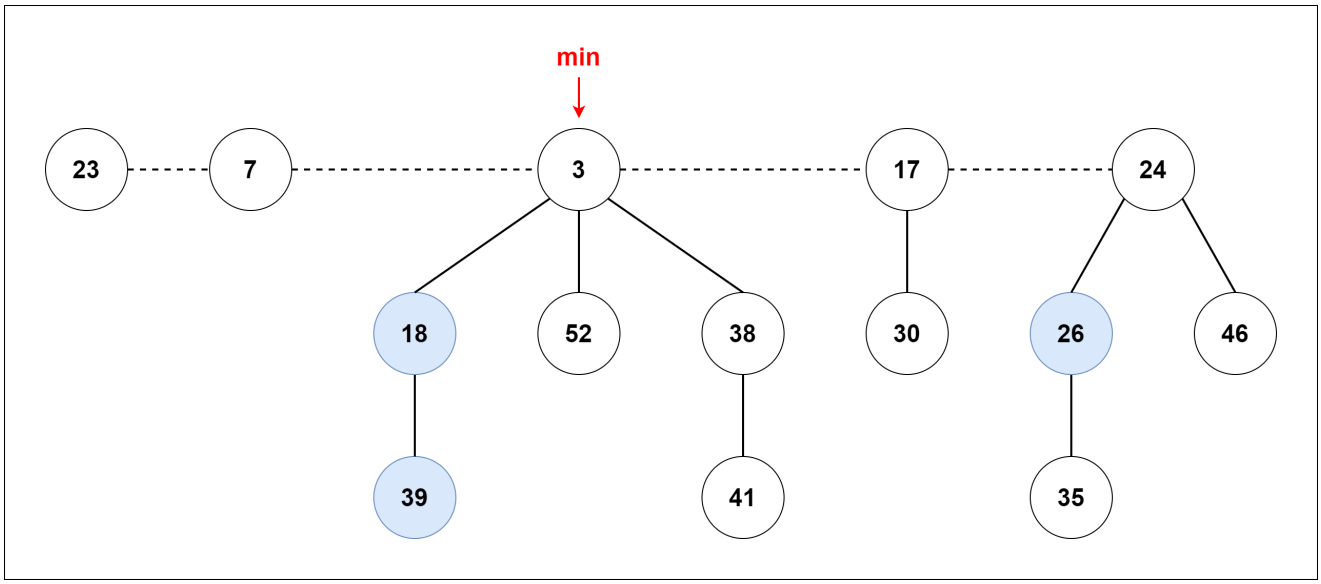
Fibonacci Heap

A Fibonacci Heap is a special way to organize data, kind of like a collection of trees. Each of these trees follows a rule, similar to what we've learned about min heaps or max heaps before. Remember, these rules help us make sure that the most important value is on top.

Now, in a Fibonacci Heap, a tree can be a little different. It's not limited to just having two children. It could have more, or sometimes none at all. And that's what makes it unique. Plus, the really cool thing is that the operations you can do on this heap are super efficient, even more than what we can do with other types of heaps like binomial or binary heaps.

Now, why's it called a "Fibonacci" heap? Well, it's because of a math thing. The way these trees are built, a tree with a certain order will have at least a specific number of nodes. And that number is related to the Fibonacci sequence, which you might have heard of in math class. So, when we say a tree of order `n` has `Fn+2` nodes, we're talking about using Fibonacci numbers to figure out how many nodes it will have. In this case it is the n + 2 th Fibonacci number.



Properties of a Fibonacci Heap

A Fibonacci Heap has some important characteristics that make it work:

1. It's like a group of trees, but they follow a special rule: in each tree, the parent node is always smaller than its children. This helps us keep the most important value at the top.
2. There's a special pointer that always points to the smallest value in the whole heap. So if you need the smallest value quickly, you know where to find it.

3. This heap also keeps track of something called "marked nodes." It's kind of like a note saying that a node's value was decreased. This is important for a certain operation called "decrease key."
4. Inside the Fibonacci Heap, the trees don't really have a specific order like we've seen before. But each tree has a starting point, called the root. This helps us keep track of all the trees in the heap.

Memory Representation of the Nodes in a Fibonacci Heap

Alright, let's understand how the nodes in a Fibonacci Heap are stored in memory:

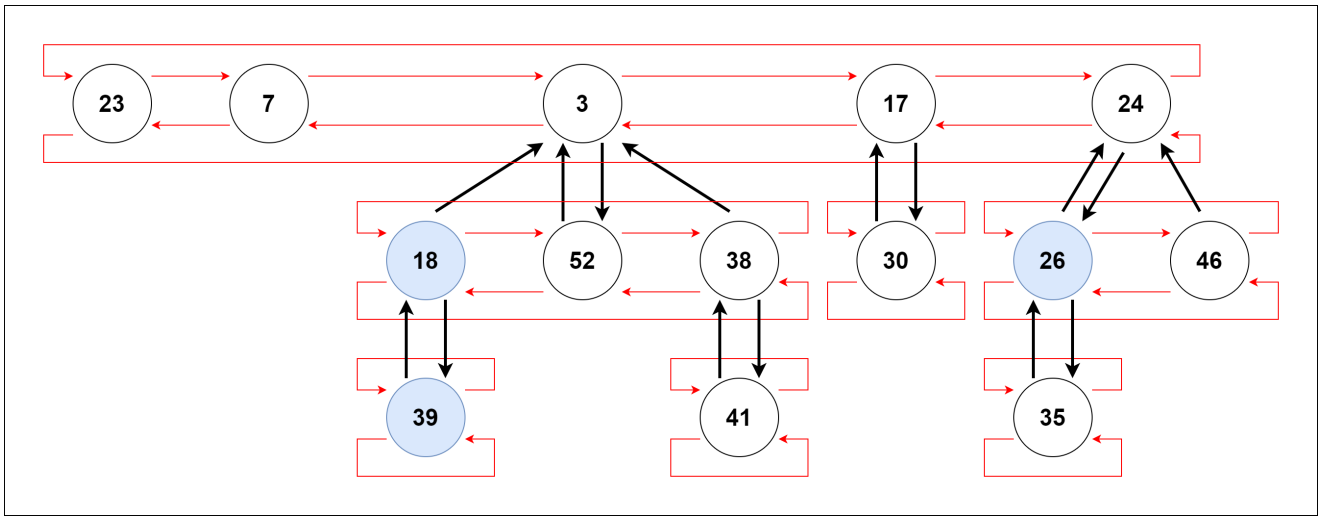
In a Fibonacci Heap, the starting points of all the trees are connected together so that we can quickly find them. Imagine it like a chain of trees.

Each parent node has its children connected to each other in a special way. It's like they're holding hands in a circle. This helps us quickly move around the children.

Using this circular linked list has two cool benefits:

1. When we want to remove a node from a tree, it's super fast and takes only a constant amount of time, like snapping your fingers.
2. Combining or joining two lists like this is also really fast, just like that, in constant time.

So, this special way of connecting nodes makes things much quicker when we're working with Fibonacci Heaps.



Operations on a Fibonacci Heap

Insertion

Alright, let's talk about how we insert elements into a Fibonacci Heap:

Here's how the process works, step by step:

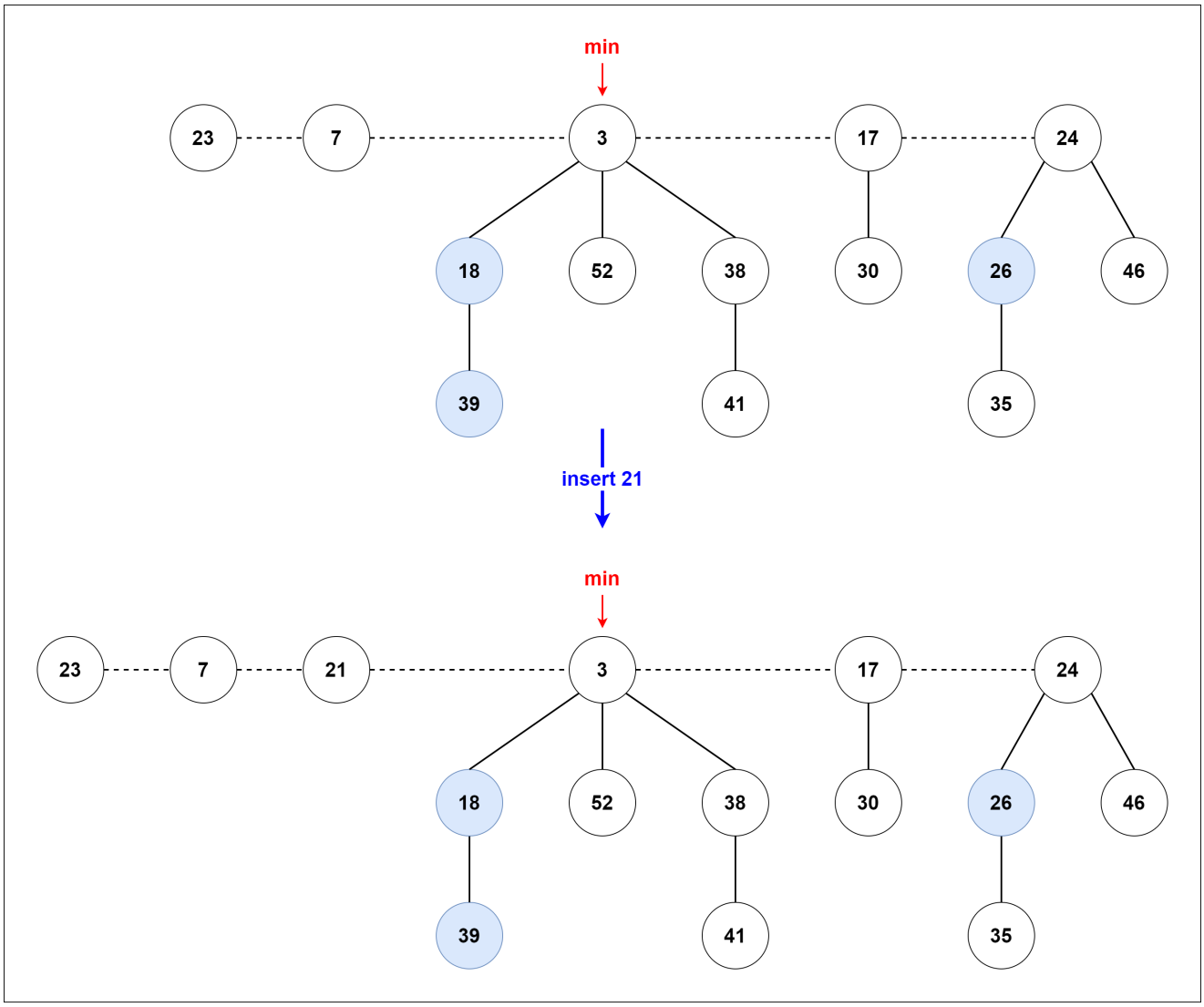
1. First, we create a new node for the element we want to insert.
2. We set some initial properties for this new node, like its degree (which is initially 0), parent (which is initially set to NIL, meaning no parent), child (also initially NIL), and its position in the circular doubly linked list (both left and right set to the node itself).
3. We also mark this new node as "not marked," to keep track of certain operations.
4. Now, we're going to put this new node into the heap. We connect it to the root list. This means it's not a child of any other node, and it's at the same level as other root nodes.
5. If the heap was empty before we inserted this node, we simply mark this new node as the minimum.
6. If the heap wasn't empty, we compare the key of this new node with the key of the current minimum node. If the new node's key is smaller, we update the minimum to be this new node.
7. Finally, we increase the total count of nodes in the heap.

Algorithm:

```
insert(H, x)
    degree[x] = 0
    p[x] = NIL
```

```
    child[x] = NIL
    left[x] = x
    right[x] = x
    mark[x] = FALSE
    concatenate the root list containing x with root list H
    if min[H] == NIL or key[x] < key[min[H]]
        then min[H] = x
    n[H] = n[H] + 1
```

So, that's how we insert a new element into a Fibonacci Heap. We make sure to update properties and connections as needed to maintain the heap's special structure and properties.



Find Min

Alright, let's talk about finding the minimum element in a Fibonacci Heap:

The process is actually quite simple. You see, there's a special pointer in the Fibonacci Heap called the "min" pointer. This pointer always points to the node with the smallest value in the

entire heap. So, when we want to find the minimum element, we just follow this pointer, and it leads us directly to the smallest element in the heap.

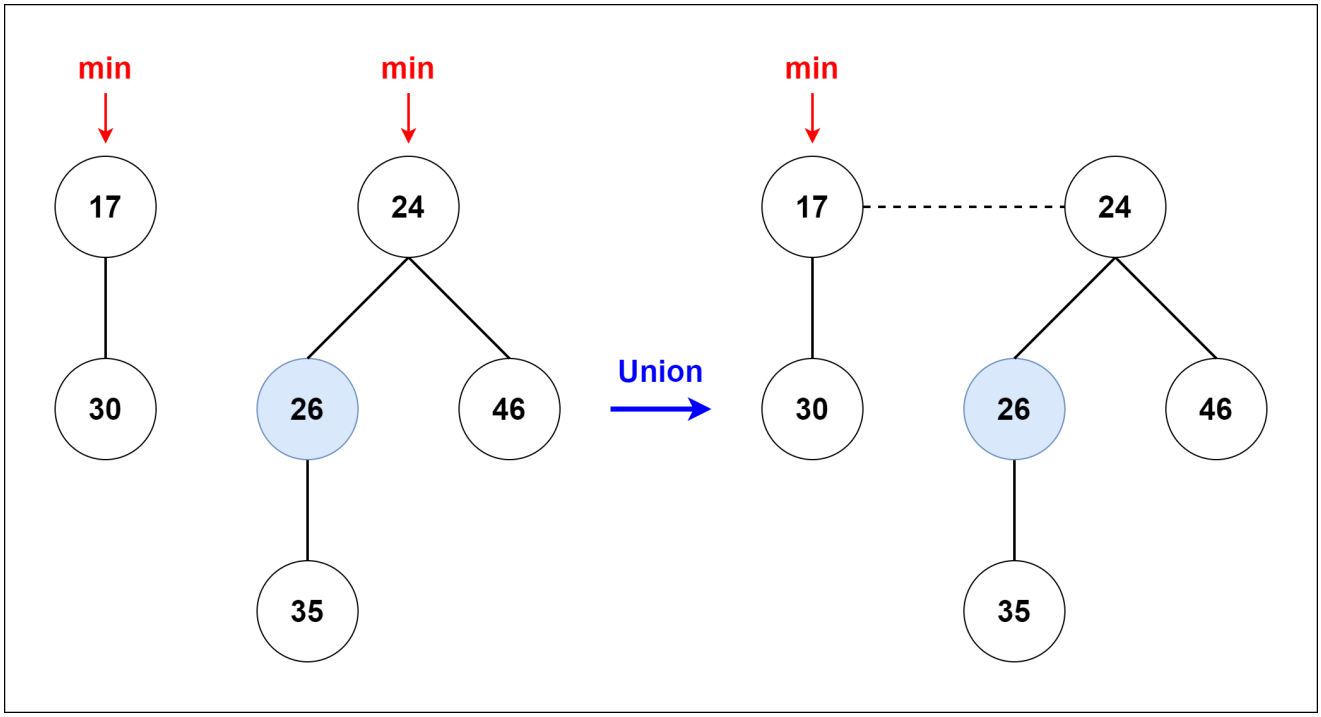
Think of it like a shortcut – instead of searching through all the nodes to find the smallest one, we can just follow the min pointer and there it is! This makes finding the minimum element a really quick operation in a Fibonacci Heap.

Union

Let's dive into the process of uniting two Fibonacci Heaps, which is like combining them into a single heap. Here's how it's done:

1. First, we take the roots of both heaps and simply put them together. It's like we're merging their root lists into one big root list.
2. Then, we go through this new root list and find the node with the smallest key. That node becomes the new minimum in the combined heap, so we update the "min" pointer to point to it.

Think of it as bringing two heaps together and arranging their roots in one line. Then we pick the smallest element from all these roots to be the new leader of the combined heap. This way, we've successfully united two heaps while ensuring that the minimum element is still easily accessible through the "min" pointer.



Extract Min

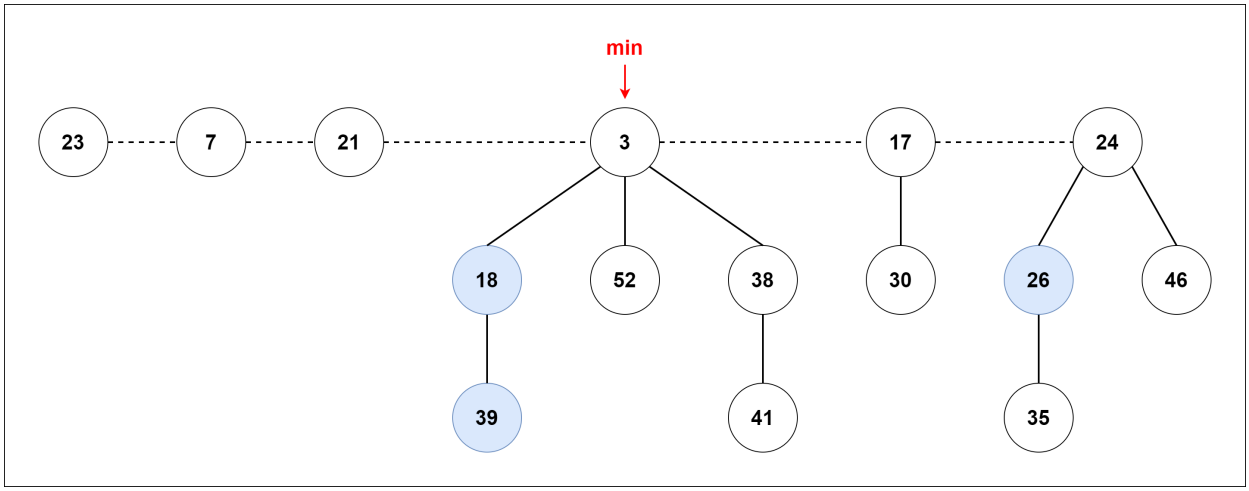
Alright, let's delve into one of the most crucial operations of a Fibonacci Heap - the Extract Min operation. This is where we remove the node with the smallest value from the heap and then readjust the structure of the tree.

Here's how we perform this operation:

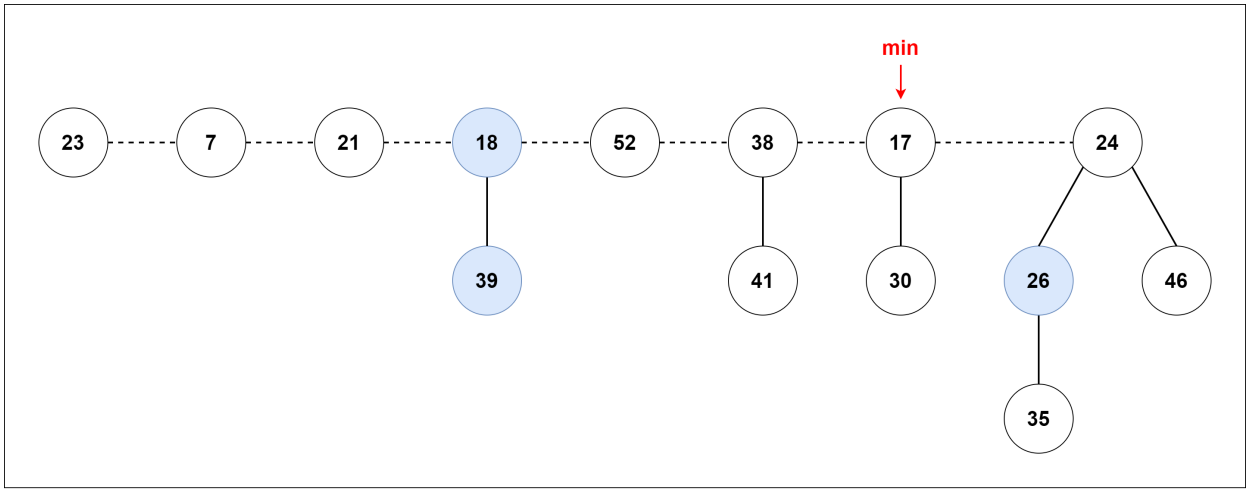
1. First, we delete the node that holds the minimum value in the heap. This is usually the root of one of the trees in the heap.
2. After that, we update the "min" pointer. This pointer now points to the next root in the root list. Remember, the root list is like a chain of trees, and this pointer helps us keep track of where the smallest element now resides.
3. Now, we do a bit of rearranging to ensure that no two trees in the root list have the same degree (number of children). We create an array that's as big as the maximum degree of any tree in the heap before deletion.
4. We then go through the root list, and for each tree, we map its degree to the corresponding slot in the array.
5. If two different trees have the same degree, we perform a union operation on those trees. This union maintains the min-heap property, which means the smallest value remains at the root.

This process might sound complex, so let's break it down with an example:

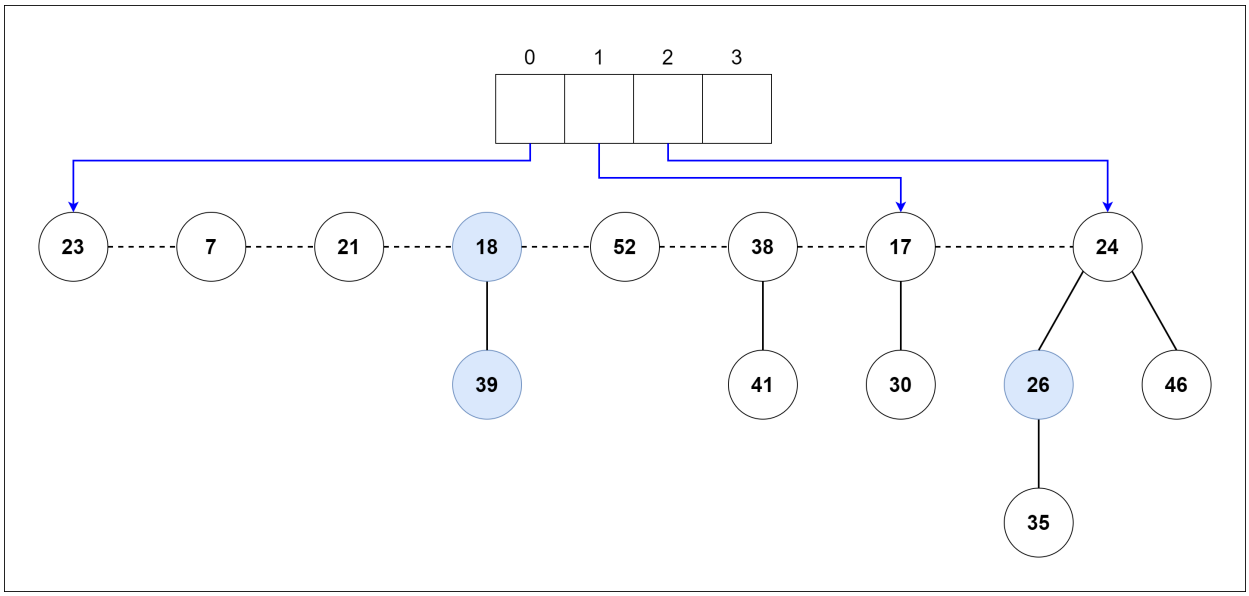
1. Imagine we have a Fibonacci Heap with various trees.



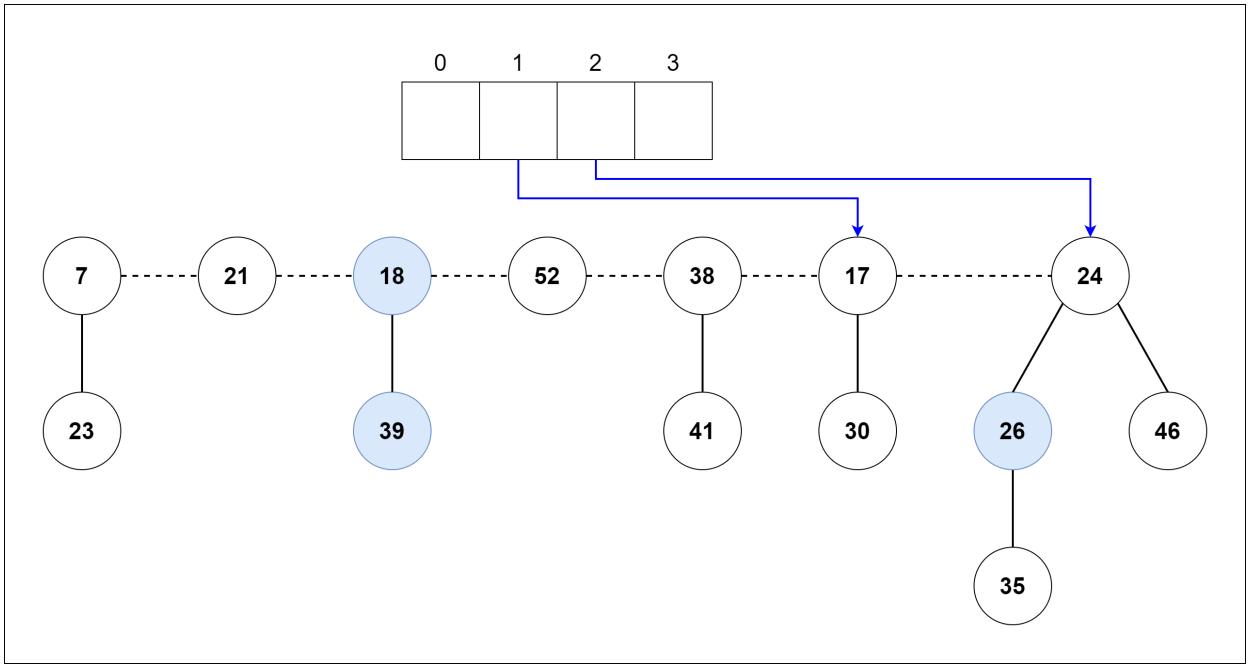
2. We start by removing the minimum node, adding its children to the root list, and updating the "min" pointer.



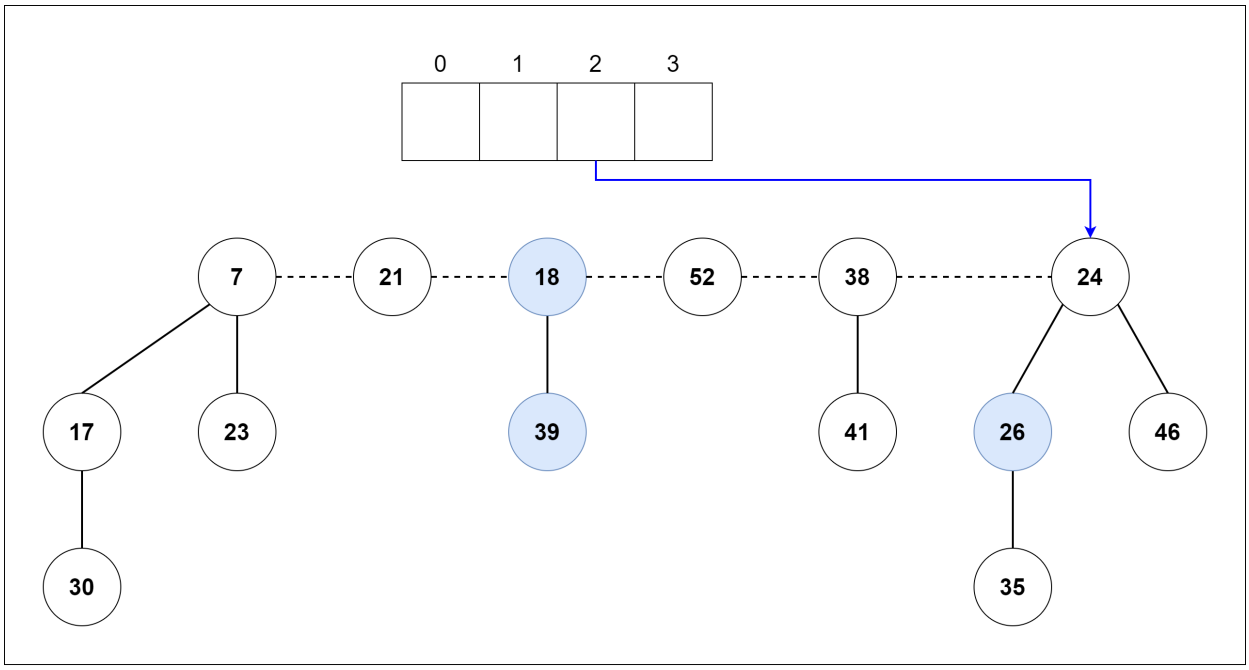
3. Let's say the maximum degree in our trees is 3. We create an array of size 4 (one extra for convenience).



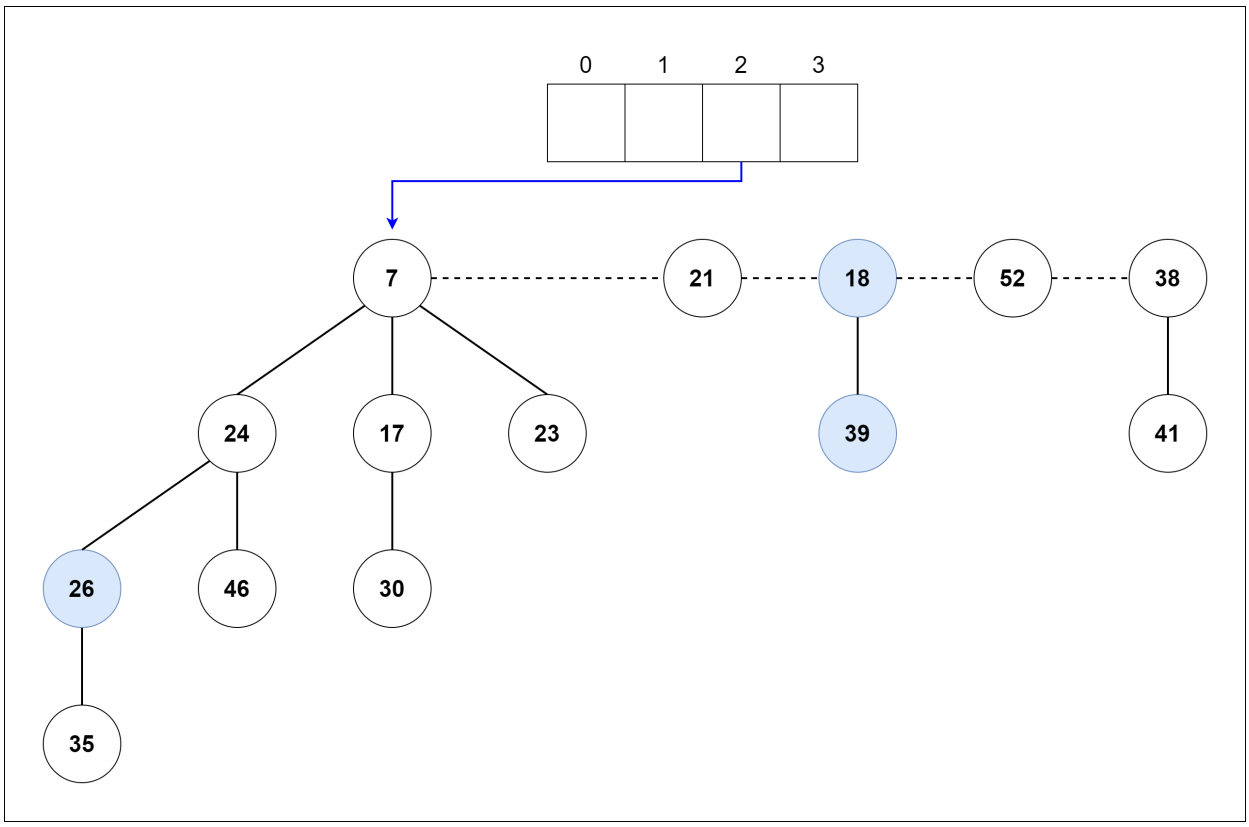
4. As we traverse the trees, we map their degrees to the array.
5. When we find trees with the same degree, like ones with 23 and 7, we combine them.



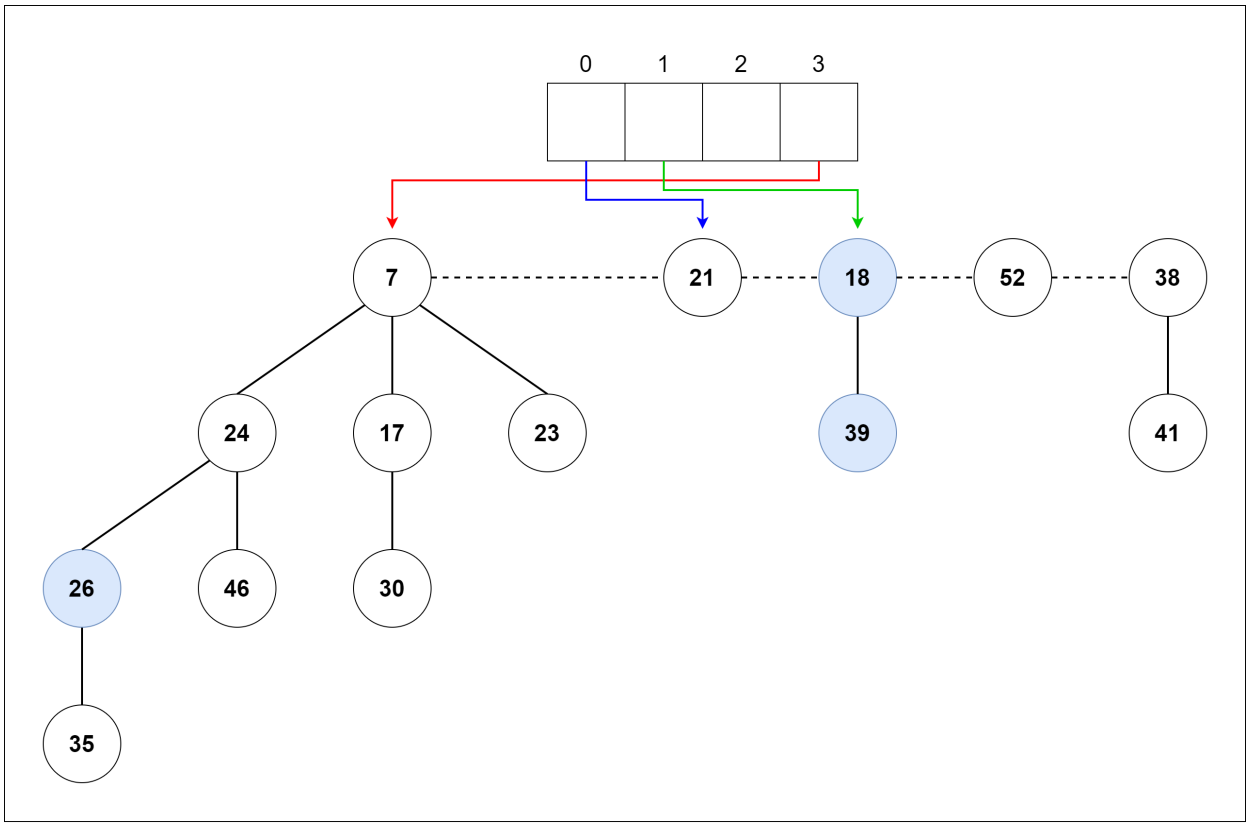
6. Again, 7 and 17 have the same degrees, so unite them as well.



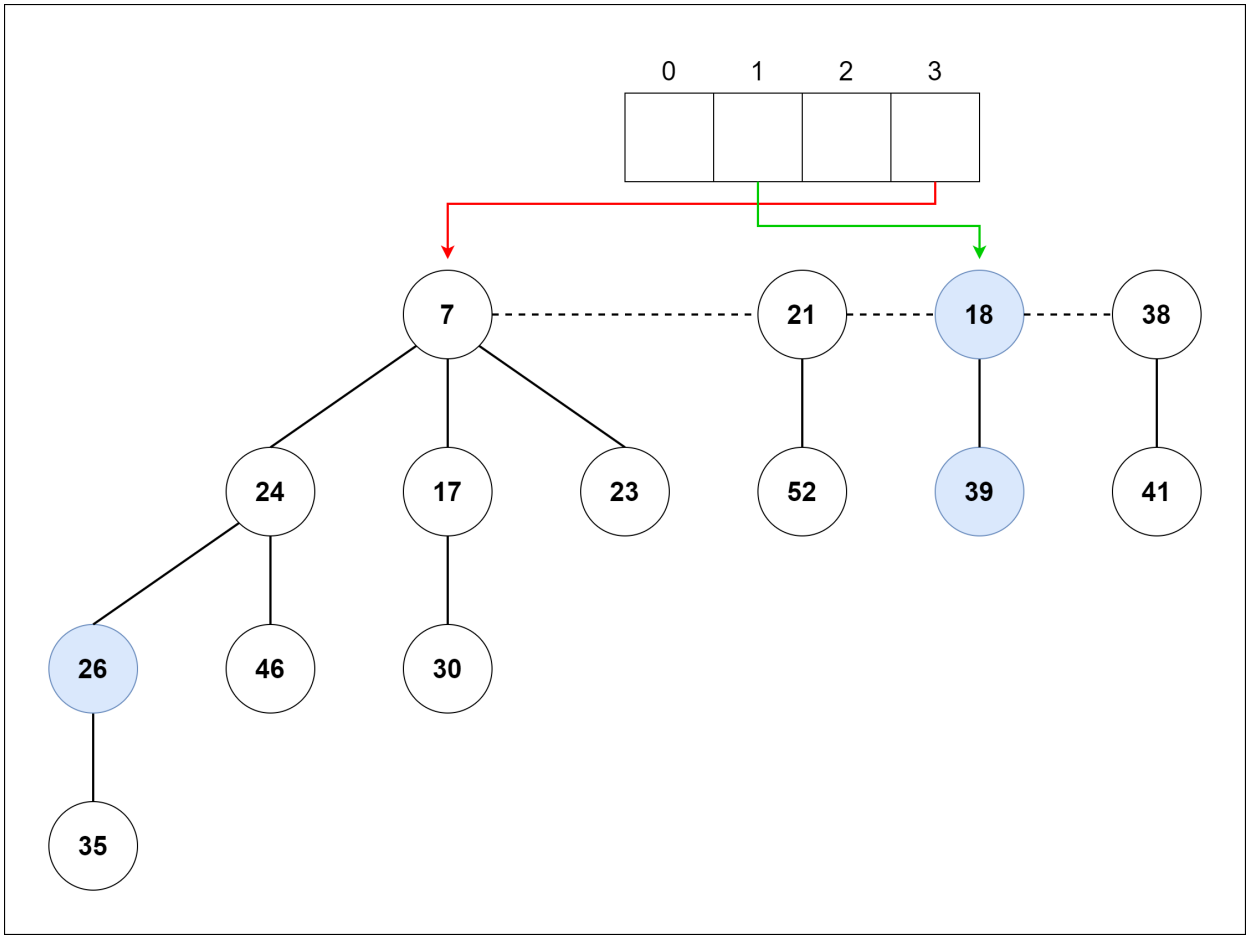
7. Again 7 and 24 have the same degree, so unite them.



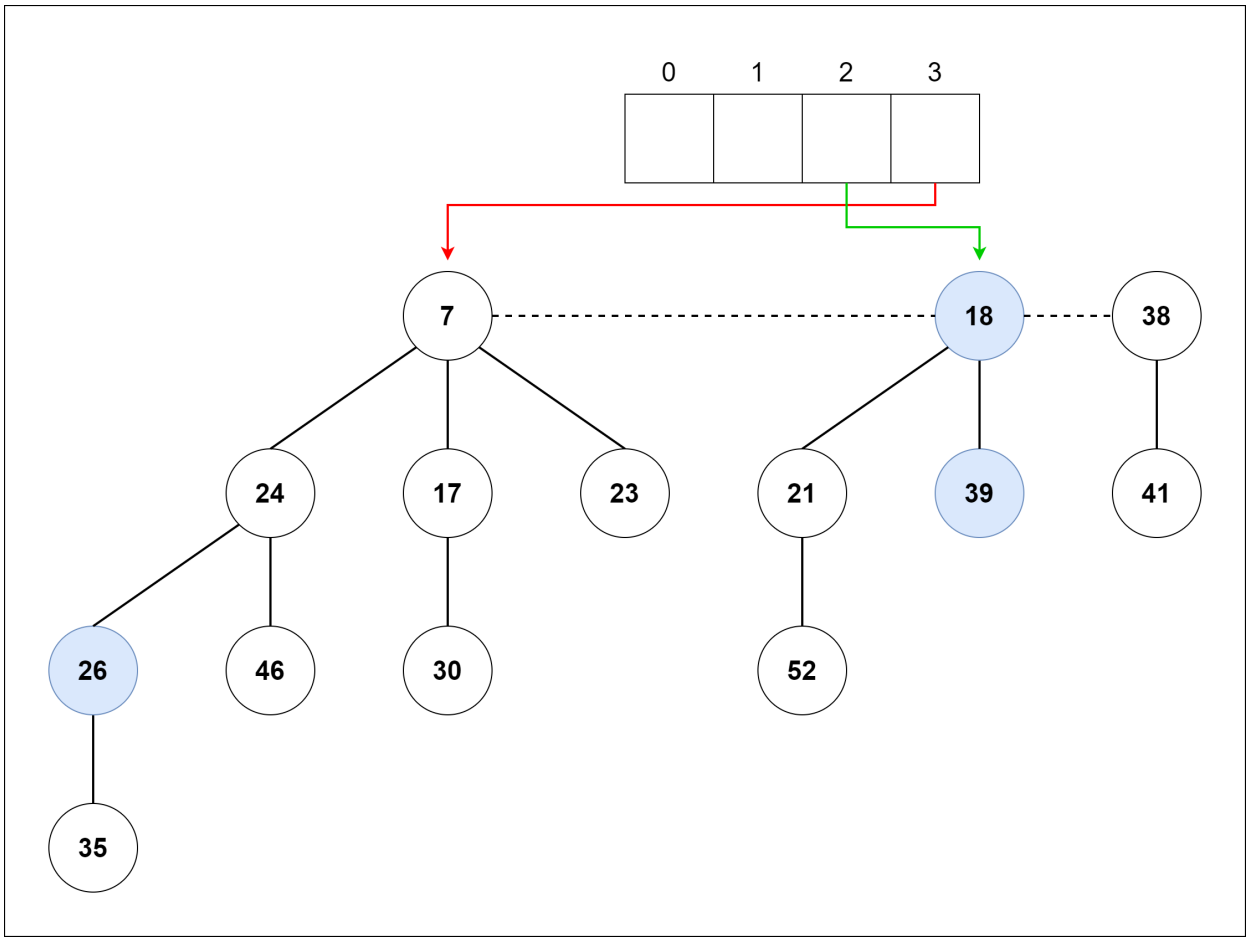
8. Map the next nodes.



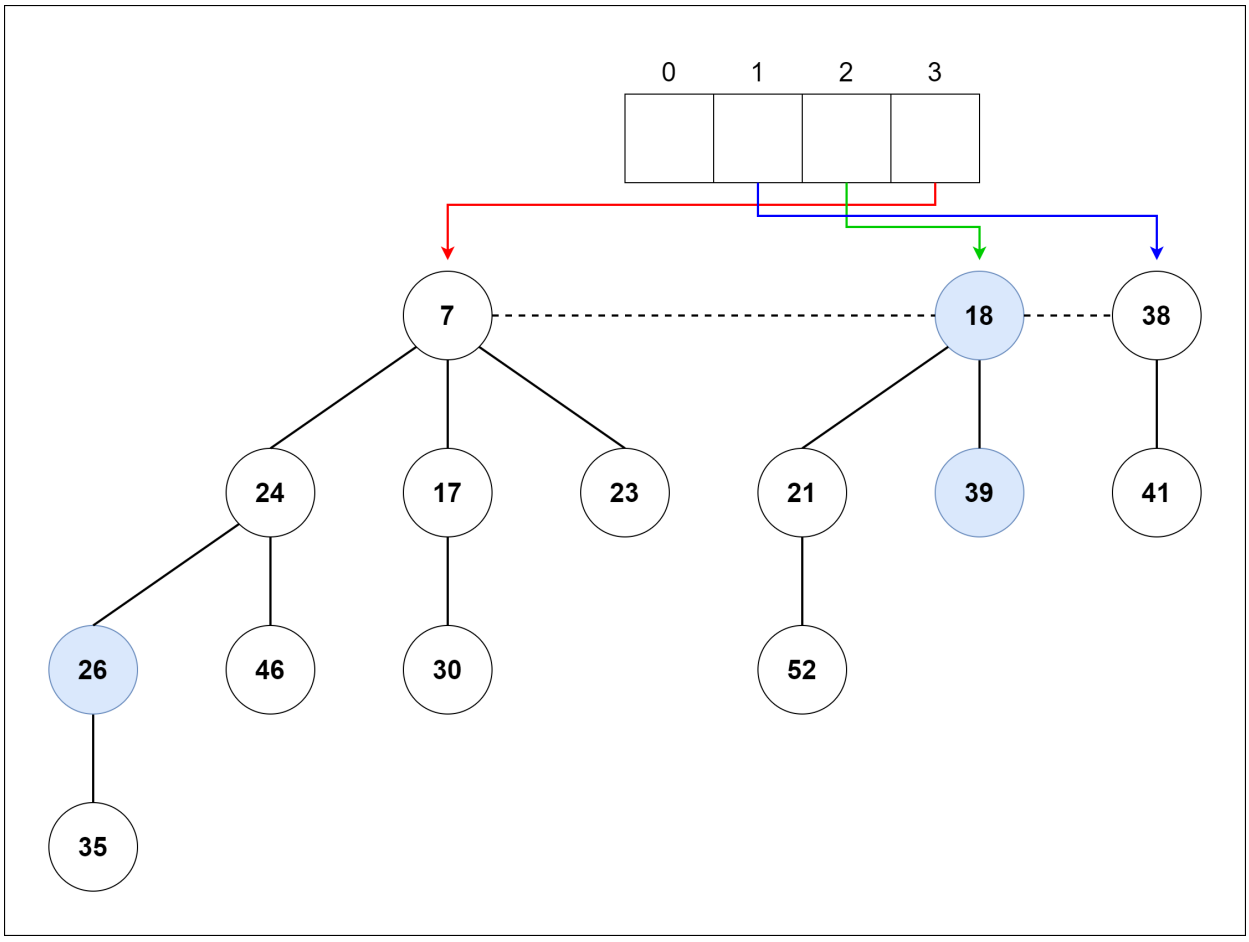
9. Again, 52 and 21 have the same degree, so unite them.



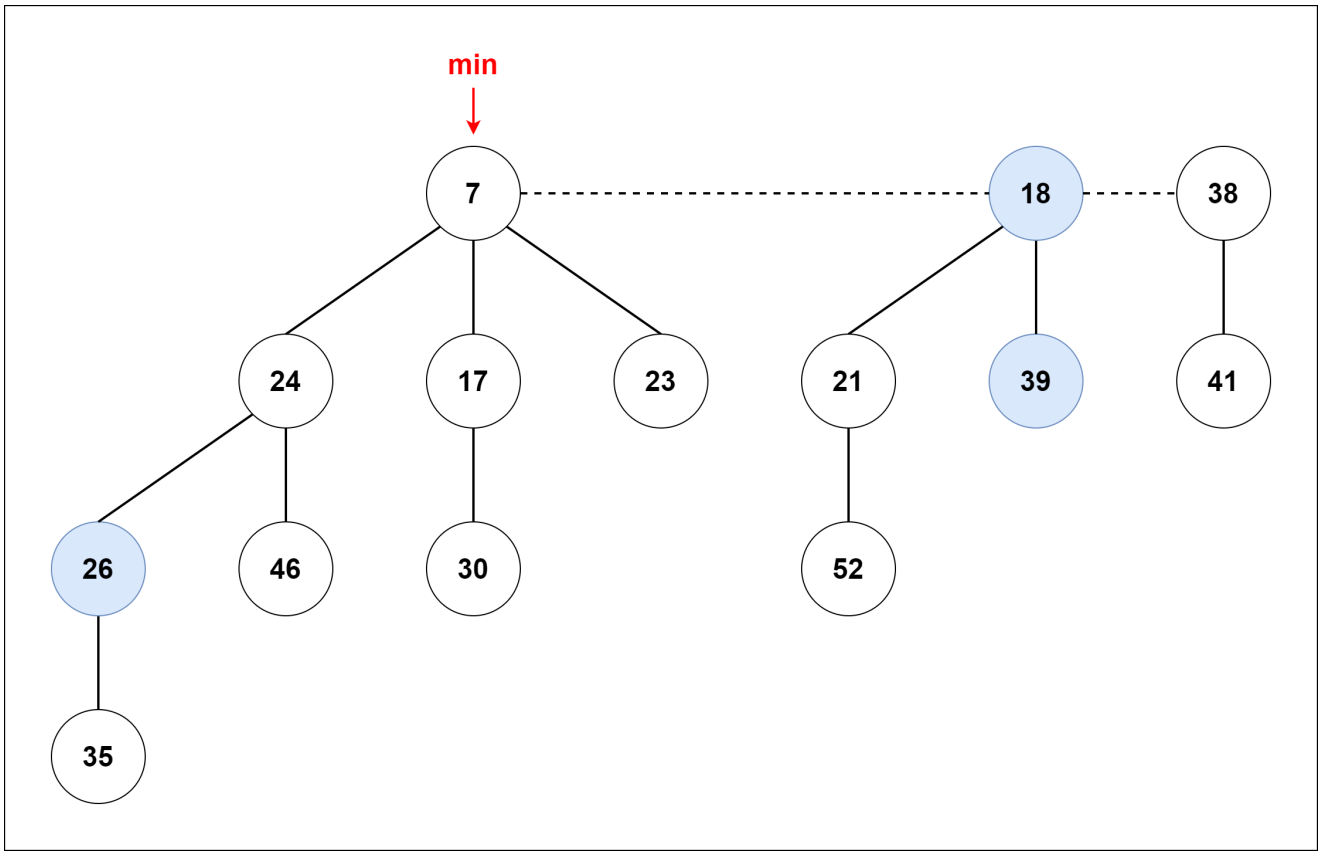
10. Similarly, unite 21 and 18.



11. Map the remaining root.



12. After these steps, the heap structure is readjusted, and we have our new heap configuration.



So, in essence, the Extract Min operation ensures that we efficiently find the smallest value, remove it from the heap, and then maintain the structure of the heap so that the min-heap property holds true. This keeps our Fibonacci Heap in good shape for further operations.

C++ Example

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

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

using namespace std;

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

  char C;
```

```
};

// Implementation of Fibonacci heap
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(); }
};

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

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

// Insert node
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;
}

// Create linking
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++;
}

// Union Operation
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;
}
```

```
// Display the heap
int FibonacciHeap::Display(node *H) {
  node *p = H;
  if (p == NULL) {
    cout << "Empty Heap" << endl;
    return 0;
  }
  cout << "Root Nodes: " << endl;

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

// Extract min
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;
}

// Consolidation Function
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];
    }
  }
}

// Decrease Key Operation
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;
}

// Cutting Function
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';
}

// Cascade cut
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);
    }
  }
}

// Search function
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;
}

// Deleting key
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(3);
  H = fh.Insert(H, p);
  p = fh.Create_node(17);
  H = fh.Insert(H, p);
  p = fh.Create_node(24);
  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

Let's look at the complexities of various operations in a Fibonacci Heap. These complexities help us understand how efficient these operations are in terms of time.

- **Insertion**: When we insert an element, it's remarkably efficient. It takes constant time, denoted as O(1), which means the time it takes doesn't grow with the size of the heap. So, adding elements to the heap is very fast.
- **Find Min**: Finding the smallest element is also super quick, taking constant time of O(1). This is because we maintain a pointer to the smallest element in the heap.
- **Union**: Combining two heaps is also very efficient, taking O(1) time. This means that merging two heaps won't take much time, no matter how big they are.
- **Extract Min**: Removing the smallest element, although efficient, takes a bit more time. It's logarithmic in nature, O(log n). This means that as the size of the heap grows, the

time taken to extract the smallest element increases, but not too drastically.

- **Decrease Key**: Reducing the value of a key takes constant time, O(1). This operation allows us to update the priority of an element quickly.
- **Delete Node**: Deleting a node also takes logarithmic time, O(log n). So, removing any node from the heap, including the smallest one, is reasonably efficient.

Operation	Time Complexity
Insertion	O(1)
Find Min	O(1)
Union	O(1)
Extract Min	O(log n)
Decrease Key	O(1)
Delete Node	O(log n)

In summary, Fibonacci Heaps are designed to perform most operations very efficiently. Operations like insertion, finding the minimum, and merging heaps are lightning fast, while more complex operations like extracting the minimum or deleting a node take a bit more time but still offer efficient performance.

Fibonacci Heap Applications

Let's discuss how Fibonacci Heaps are used in real-world scenarios.

One significant application is in improving the performance of an algorithm called Dijkstra's algorithm. This algorithm is used to find the shortest path in a graph, which has various practical applications like finding the quickest route between locations on a map. The unique properties of Fibonacci Heaps make them useful in optimizing the time complexity of this algorithm, ultimately leading to faster and more efficient pathfinding.