# Advanced data structures and Applications

Pham Quang Dung and Do Phan Thuan

Computer Science Department, SoICT,
Hanoi University of Science and Technology.

August 15, 2016

# Outline

- Priority queues
- Disjoint Sets
- Applications
  - Dijkstra algorithm
  - Kruskal algorithm
  - Prim algorithm

# Priority queue

- Data structure stores a set of elements, each of which is associated with a key with following operations:
  - Insert an element
  - Return an element with a minimum key
  - Return an element with a minimum key and remove this element from the queue
  - Decrease the key of an element
- Applications
  - Dijkstra algorithm for finding shortest paths in a graph
  - Prim algorithm for finding a minimum spanning tree of a graph
  - Heap sort
  - ...

# Binary heaps

- Complete binary tree
  - All levels, except possibly the last one, are fully filled
  - If the last level is not fully filled, the nodes of that level are filled from left to right
- Heap property
  - The key of every node is less than or equal to the keys of its children
  - Min element is the root
  - Heap with $n$ elements has height $\lfloor logn \rfloor$

# Binary heaps - implementation

- Use an array $x[1..n]$
  - $parent(x[i]) = x[(i)/2]$
  - $leftChild(x[i]) = x[2i]$
  - $rightChild(x[i]) = x[2i + 1]$
- Insert, Decrease-Key, Extract-Min: $\mathcal{O}(logn)$
- Find-Min: $\mathcal{O}(1)$

# Binary heaps - implementation

**Algorithm 1:** Heapify($x[1..n], k$)

$t \leftarrow x[k]$;
**while** $k < n$ **do**
  $c \leftarrow 2 \times k$;
  **if** $c < n \wedge x[c+1] < x[c]$ **then**
    $c \leftarrow c + 1$;
  **if** $c < n \wedge t > x[c]$ **then**
    $x[k] \leftarrow x[c]$;
  **else**
    BREAK;
  $k \leftarrow c$;
$x[k] \leftarrow t$;

# Binary heaps - implementation

**Algorithm 2:** BuildMinHeap($x[1..n]$)

$k \leftarrow n/2$;
**while** $k > 0$ **do**
    Heapify($x[1..n], k$);
    $k \leftarrow k - 1$;

# Binary heaps - implementation

**Algorithm 3:** ExtractMin($x[1..n]$)

**if** $n$=*0* **then**
  **return** *NULL*;

$r \leftarrow x[1]$;
$x[1] \leftarrow x[n]$;
$n \leftarrow n - 1$;
Heapify($x[1..n], 1$);
**return** $r$;

# Binary heaps - implementation

**Algorithm 4:** DecreaseKey($x[1..n], k$)

```
t ← x[k];
while k > 1 do
    p ← k/2;
    if t < x[p] then
        x[k] ← x[p];
    else
        BREAK;
    k ← p;
x[k] ← t;
```

# Binary heaps - implementation

```
package week12;
import java.util.*;

public class MinHeap<AnyType extends Comparable<AnyType>> {
  private int sz;
  private AnyType[] arr;// elements are indexed from 1, 2, ... (do not
  private HashMap<AnyType, Integer> mapIndex;//  map an element to its

  public MinHeap() {
    sz = 0;
    arr = (AnyType[]) new Comparable[10];
    mapIndex = new HashMap<AnyType, Integer>();
  }

  public MinHeap(AnyType[] L) {
    sz = L.length;
    arr = (AnyType[]) new Comparable[L.length + 1];
    System.arraycopy(L, 0, arr, 1, L.length);
    for(int i = 1; i <= L.length; i++)
      mapIndex.put(arr[i], i);
    buildHeap();
  }
  . . .
}
```

# Binary heaps - implementation

```java
  public boolean empty(){
    return sz <= 0;
  }
  private void scale() {
    AnyType[] tmp = arr;
    arr = (AnyType[]) new Comparable[arr.length * 2];
    System.arraycopy(tmp, 1, arr, 1, sz);
    for(int i = 1; i <= sz; i++)
      mapIndex.put(arr[i], i);
  }
  private void buildHeap() {
    for (int k = sz / 2; k > 0; k--) {
      heapify(k);
    }
  }
  private void swap(int a, int b){
    AnyType tmp = arr[a]; arr[a] = arr[b]; arr[b] = tmp;
    mapIndex.put(arr[a], a);
    mapIndex.put(arr[b], b);
  }
  . . .
}
```

# Binary heaps - implementation

```
public void decreaseKey(AnyType e){
  int k = mapIndex.get(e);
  AnyType tmp = arr[k];
  int parent;
  for(;k > 1; k = parent){
    parent = k/2;
    if(tmp.compareTo(arr[parent]) < 0){
      arr[k] = arr[parent];
      mapIndex.put(arr[k], k);
    }else break;

  }
  arr[k] = tmp;
  mapIndex.put(arr[k], k);
}
. . .
}
```

# Binary heaps - implementation

```
  private void heapify(int k) {
    AnyType tmp = arr[k];
    int child;

    for (; 2 * k <= sz; k = child) {
      child = 2 * k;

      if (child < sz && arr[child].compareTo(arr[child + 1]) > 0)
        child++;

      if (tmp.compareTo(arr[child]) > 0){
        arr[k] = arr[child];
        mapIndex.put(arr[k], k);
      }else
        break;
    }
    arr[k] = tmp;
    mapIndex.put(arr[k], k);
  }
  . . .
}
```

# Binary heaps - implementation

```
  public void sort(AnyType[] L) {
    sz = L.length;
    arr = (AnyType[]) new Comparable[sz + 1];
    System.arraycopy(L, 0, arr, 1, sz);
    for(int i = 1; i <= sz; i++) mapIndex.put(arr[i], i);

    buildHeap();
    for (int i = sz; i > 0; i--) {
      swap(i,1);
      sz--;
      heapify(1);
    }
    for (int k = 0; k < arr.length - 1; k++)
      L[k] = arr[arr.length - 1 - k];
  }
  public boolean contains(AnyType a){
    return mapIndex.get(a) != null;
  }
  . . .
}
```

# Binary heaps - implementation

```
  public AnyType deleteMin(){
    if (sz == 0) return null;
    AnyType min = arr[1];
    arr[1] = arr[sz--];
    mapIndex.put(arr[1],1);
    heapify(1);
    return min;
  }

  public void insert(AnyType x) {
    if (sz == arr.length - 1) scale();
    sz++;
    int i = sz;
    for (; i > 1 && x.compareTo(arr[i / 2]) < 0; i = i / 2){
      arr[i] = arr[i / 2];
      mapIndex.put(arr[i], i);
    }
    arr[i] = x;
    mapIndex.put(arr[i], i);
  }
}
```

# Fibonacci heap

- Collection of rooted trees (**min-heap ordered**)
- Each node $x$
  - $p(x)$: parent of $x$
  - $child(x)$: points to one of the children of $x$
  - Children of $x$ are linked together in a circular (doubly linked list, called child list of $x$)
  - $left(x)$ and $right(x)$: pointers to the left and right siblings of $x$
  - $degree(x)$: the number of children of $x$
  - $mark(x)$: TRUE if $x$ has lost a child since the last time $x$ was made the child of another nodes
    - ⋆ A newly created node $y$ is unmarked: $mark(y) = \text{FALSE}$
    - ⋆ A node $y$ becomes unmarked whenever it is made the child of another node
    - ⋆ $mark(.)$ attribute is set to FALSE when we consider the DECREASE-KEY operations

# Fibonacci heap

- For each fibonacci heap $H$
  - The roots of all the trees are linked together into a doubly linked list, called the **root list** of the fibonacci heap
  - Trees may appear in any order within the root list
  - $min(H)$ is a pointer to the root of a tree containing the minimum key (called **minimum node** of the fibonacci heap)
  - $n(H)$: the number of nodes of $H$
  - $t(H)$: number of trees in the root list of $H$
  - $m(H)$: number of marked nodes of $H$
  - Potential function: $\Phi(H) = t(H) + 2m(H)$
- We denote $D(n)$ the maximum degree of any node in a $n$-node fibonacci heap
- It will be proved that $D(n) = \mathcal{O}(lgn)$

# Fibonacci heap - Implementing operations

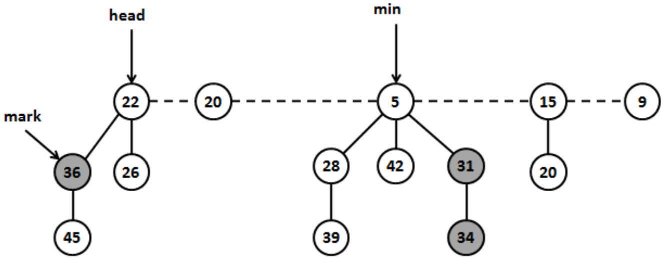
- Create an empty heap
- $\mathcal{O}(1)$

**Algorithm 5:** FIB-HEAP-MAKE()

$n(H) \leftarrow 0;$
$min(H) \leftarrow \text{NIL};$
$t(H) \leftarrow 0;$
$m(H) \leftarrow 0;$
**return** $H$;

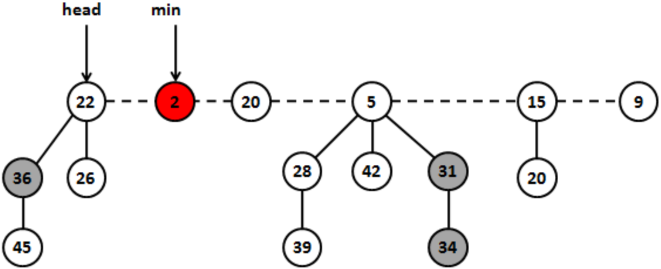
# Fibonacci heap - Implementing operations

Current heap



# Fibonacci heap - Implementing operations

Insert node with key = 2



# Fibonacci heap - Implementing operations

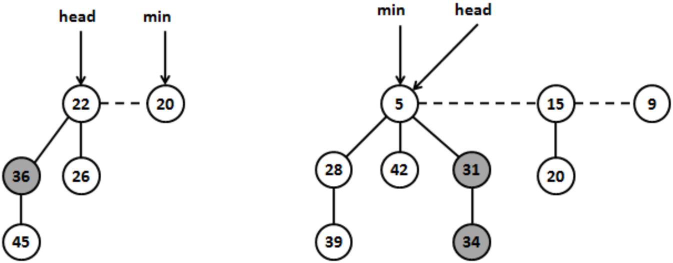
- Insert a node $x$ to the heap $H$
- No consolidation as in binomial heap (lazzy)
- Analysis ($H'$ is the resulting heap)
  - Increase in potential is $t(H') + 2m(H') - (t(H) + 2m(H)) = t(H) + 1 + 2m(H) - (t(H) + 2m(H)) = 1$
  - $\Rightarrow$ Amortized cost if $\mathcal{O}(1) + 1 = \mathcal{O}(1)$ (because the actual cost is $\mathcal{O}(1)$)

**Algorithm 6:** FIB-HEAP-INSERT($H, x$)

$degree(x) \leftarrow 0$;
$p(x) \leftarrow$ NIL;
$child(x) \leftarrow$ NIL;
$mark(x) \leftarrow$ FALSE;
**if** $min(H) = NIL$ **then**
  Create a root list for $H$ containing only $x$;
  $min(H) \leftarrow x$;
**else**
  Insert $x$ into $H$'s root list;
  **if** $key(x) < key(min(H))$ **then**
    $min(H) \leftarrow x$;
  $n(H) \leftarrow n(H) + 1$;

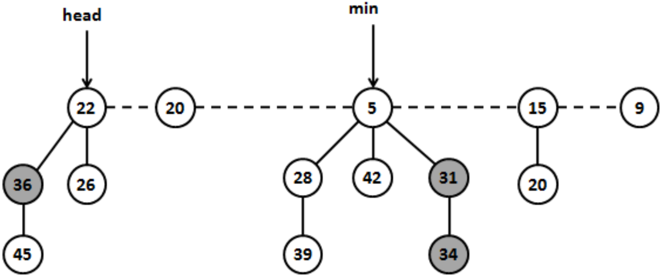
# Fibonacci heap - Implementing operations

Two fibonacci heaps to be unified



# Fibonacci heap - Implementing operations

After union



# Fibonacci heap - Implementing operations

- Unify two heaps
- Analysis
  - Change in potential is $\Phi(H) - \Phi(H_1) - \Phi(H_2) = 0$
  - Since actual cost is $\mathcal{O}(1) \Rightarrow$ amortized cost is $\mathcal{O}(1)$

---

**Algorithm 7:** FIB-HEAP-UNION($H_1, H_2$)

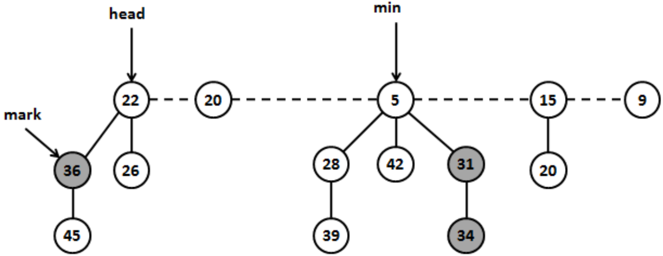
---

$H \leftarrow$ FIB-HEAP-MAKE();
$min(H) \leftarrow min(H_1)$;
Concatenate the root list of $H_2$ with the root list of $H_1$;
**if** $min(H_1) = NIL$ *or* $min(H_2) \neq NIL$ *and* $key(min(H_2)) < key(min(H_1))$ **then**
  $min(H) \leftarrow min(H_2)$;
$n(H) \leftarrow n(H_1) + n(H_2)$;
**return** $H$;

---

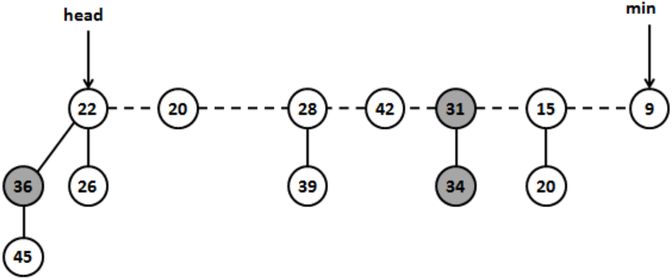
# Fibonacci heap - Implementing operations

ExtractMin



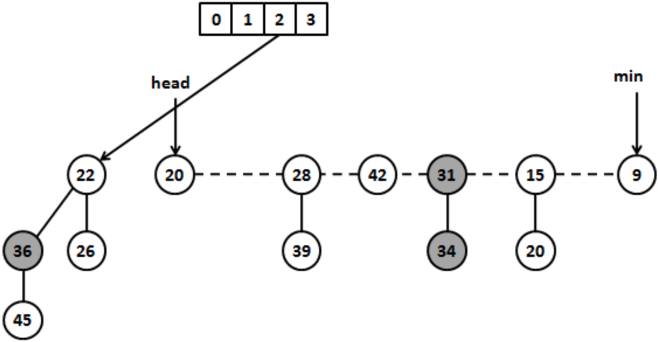
# Fibonacci heap - Implementing operations

ExtractMin: Consolidate - step 1



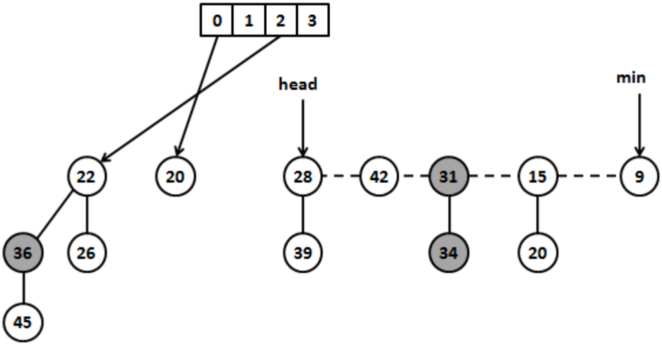
# Fibonacci heap - Implementing operations

ExtractMin: Consolidate - step 2



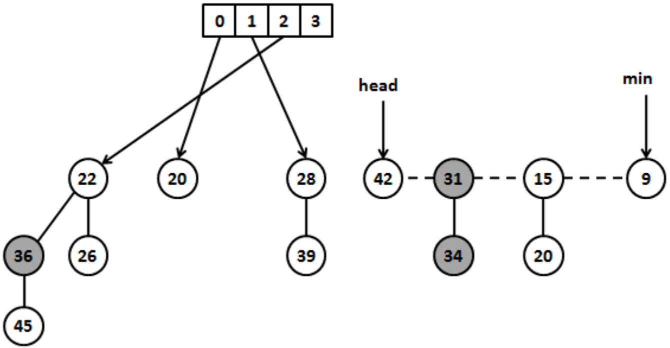
# Fibonacci heap - Implementing operations

ExtractMin: Consolidate - step 3



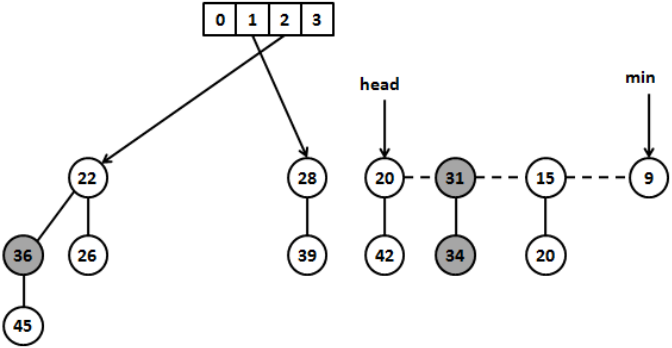
# Fibonacci heap - Implementing operations

ExtractMin: Consolidate - step 4



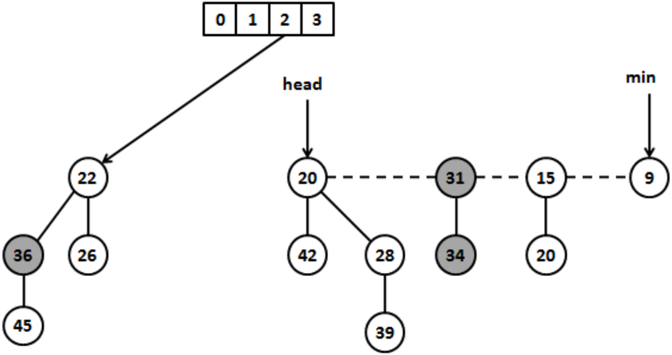
# Fibonacci heap - Implementing operations

ExtractMin: Consolidate - step 5



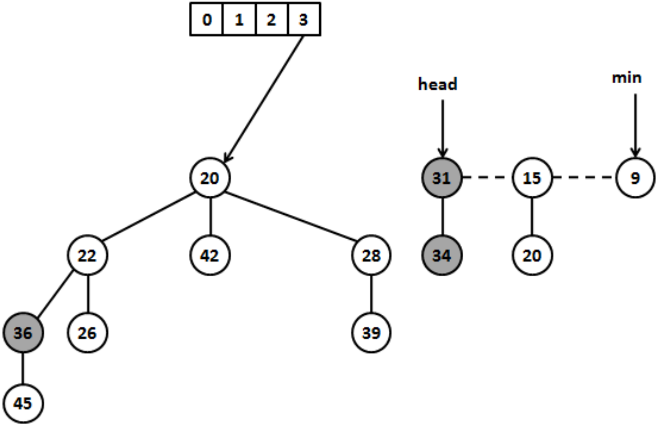
# Fibonacci heap - Implementing operations

ExtractMin: Consolidate - step 6



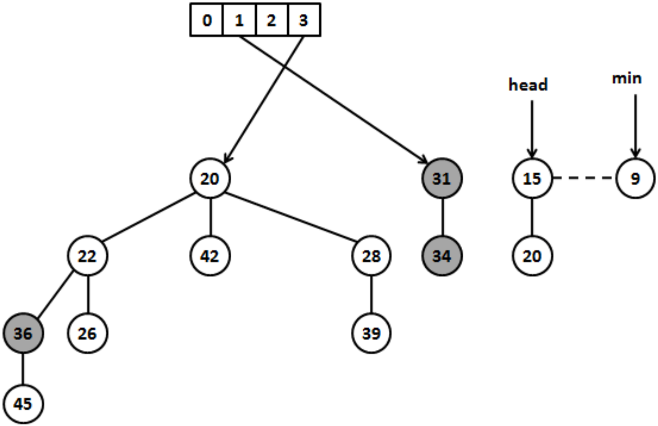
# Fibonacci heap - Implementing operations

ExtractMin: Consolidate - step 7



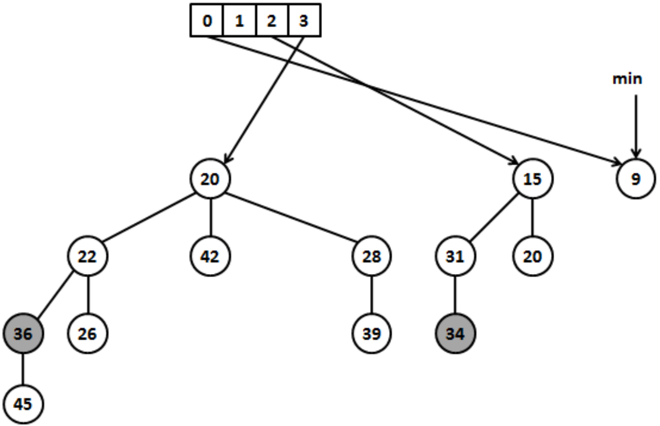
# Fibonacci heap - Implementing operations

ExtractMin: Consolidate - step 8



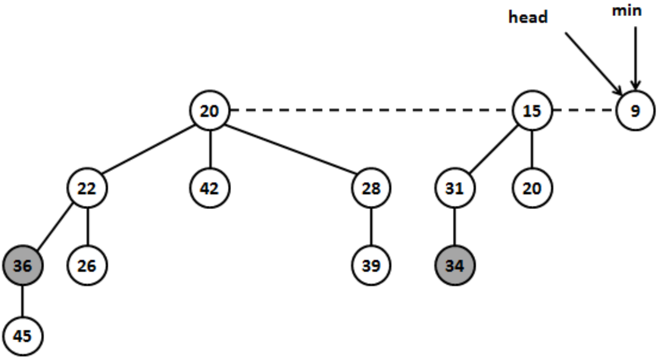
# Fibonacci heap - Implementing operations

ExtractMin: Consolidate - step 9



# Fibonacci heap - Implementing operations

ExtractMin: Consolidate - step 10



# Fibonacci heap - Implementing operations

- Extracting min element (most complicated operation)
- Delayed work of consolidating trees in the root list finally occurs
  - Linking the roots of equal degree until at most one root remains of each degree

**Algorithm 8:** FIB-HEAP-EXTRACT-MIN($H$)

$z \leftarrow min(H)$;
**if** $z \neq NIL$ **then**
    **foreach** *child* $x$ *of* $z$ **do**
        Add $x$ to the root list of $H$;
        $p(x) \leftarrow$ NIL;
    Remove $z$ from the root list of $H$;
    **if** $z = right(z)$ **then**
        $min(H) \leftarrow$ NIL;
    **else**
        $min(H) \leftarrow right(z)$;
        CONSOLIDATE($H$);
    $n(H) \leftarrow n(H) - 1$;

# Fibonacci heap - Implementing operations

**Algorithm 9:** CONSOLIDATE($H$)

```
Let A[0..D(n(H))] be a new array;
foreach i ∈ {0, ..., D(n(H))} do
    A[i] ← NIL;
foreach node w in the root list of H do
    x ← w;
    d ← degree(x);
    while A[d] ≠ NIL do
        y ← A[d];
        if key(x) > key(y) then
            exchange x with y;
        FIB-HEAP-LINK(H, y, x);
        A[d] ← NIL;
        d ← d + 1;
    A[d] ← x;
min(H) ← NIL;
foreach i ∈ {0, ..., D(n(H))} do
    if A[i] ≠ NIL then
        if min(H) = NIL then
            Create a root list for H containing only A[i];
            min(H) ← A[i];
        else
            Insert A[i] into the root list of H;
            if key(A[i]) < key(min(H)) then
                min(H) ← A[i];
```

**Algorithm 10:** FIB-HEAP-LINK($H, y, x$)

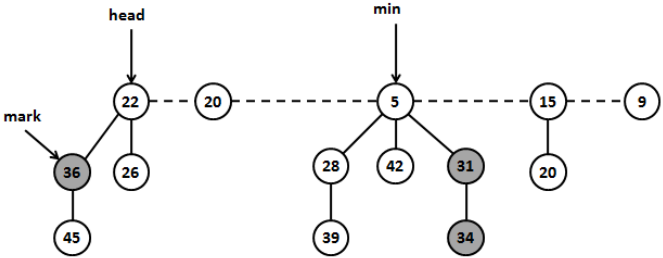
Remove $y$ from the root list of $H$;
Make $y$ a child of $x$;
$degree(x) \leftarrow degree(x) + 1$;
$mark(y) \leftarrow$ FALSE;

# Fibonacci heap - Implementing operations

- Analysis of CONSOLIDATE
  - The size of root list upon calling CONSOLIDATE is at most $\mathcal{O}(D(n)) + t(H) - 1$, since it consists of $t(H)$ nodes of the root list, minus the extracted node, plus the children of the extracted node which is $\mathcal{O}(D(n))$
  - **for** loop of lines 4–14
    - Every time through the **while** loop of lines 7–13, one of the roots is linked to another
    - $\Rightarrow$ Total amount of work is proportional to $D(n) + t(H)$
  - **for** loop of lines 16–24: $\mathcal{O}(D(n))$
- $\Rightarrow$ EXTRACT-MIN actually takes $\mathcal{O}(D(n) + t(H))$
- Potential before EXTRACT-MIN is $t(H) + 2m(H)$ and afterward is at most $D(n) + 1 + 2m(H)$, since at most $D(n) + 1$ roots remain and no nodes become marked during the operation
- Amortized cost is $\mathcal{O}(D(n) + t(H)) + (D(n) + 1 + 2m(H)) - (t(H) + 2m(H)) = \mathcal{O}(D(n))$

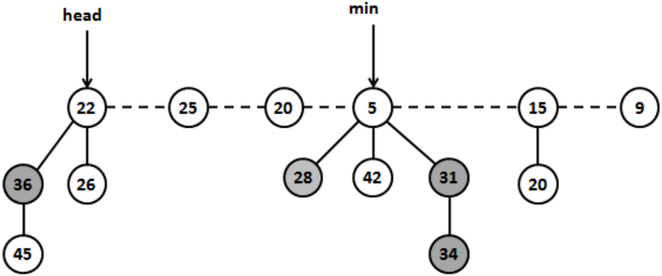
# Fibonacci heap - Implementing operations

Current heap



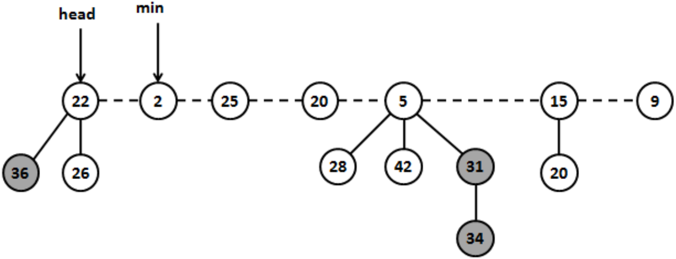
# Fibonacci heap - Implementing operations

DecreaseKey of 39 to 25

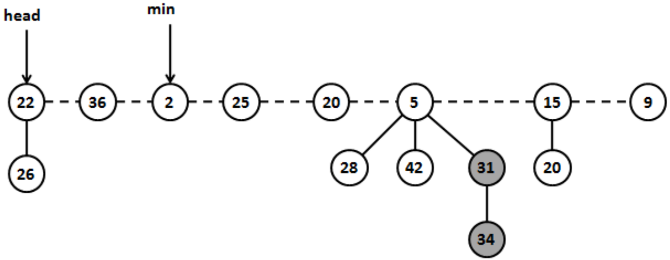


# Fibonacci heap - Implementing operations

DecreaseKey of 45 to 2



DecreaseKey of 45 to 2



# Fibonacci heap - Implementing operations

**Algorithm 11:** FIB-HEAP-DECREASE-KEY$(H, x, k)$

**if** $key(x) < k$ **then**
  error;

$key(x) \leftarrow k$;
$y \leftarrow p(x)$;
**if** $y \neq NIL$ *and* $key(x) < key(y)$ **then**
  CUT$(H, x, y)$;
  CASCADING-CUT$(H, y)$;

**if** $key(x) < key(min(H))$ **then**
  $min(H) \leftarrow x$;

**Algorithm 12:** FIB-HEAP-DELETE$(H, x)$

FIB-HEAP-DECREASE-KEY$(H, x, -\infty)$;
FIB-HEAP-EXTRACT-MIN$(H)$;

# Fibonacci heap - Implementing operations

**Algorithm 13:** CUT$(H, x, y)$

Remove $x$ from the child list of $y$;
$degree(y) \leftarrow degree(y) - 1$;
Add $x$ to the root list of $H$;
$p(x) \leftarrow$ NIL;
$mark(x) \leftarrow$ FALSE;

**Algorithm 14:** CASCADING-CUT$(H, y)$

```
z ← p(y);
if z ≠ NIL then
    if mark(y) = FALSE then
        mark(y) ← TRUE;
    else
        CUT(H, y, z);
        CASCADING-CUT(H, z);
```

# Fibonacci heap - Implementing operations

- Analysis of FIB-HEAP-DECREASE-KEY
  - Suppose that CASCADING-CUT is recursively called $c$ times from a given invocation of FIB-HEAP-DECREASE-KEY
  - Each call of CASCADING-CUT takes $\mathcal{O}(1)$ exclusive of recursive call
  - Thus, the actual cost of FIB-HEAP-DECREASE-KEY is $\mathcal{O}(c)$
  - Change in potential
    - ★ Each recursive call of CASCADING-CUT, except for the last one, cuts a marked node and clear the mark bit.
    - ★ Afterward, there are $t(H) + c$ trees (the original $t(H)$ trees, $c - 1$ trees produces by CASCADING-CUT and the tree rooted at $x$) and at most $m(H) + c - 2$ marked nodes ($c - 1$ were unmarked by CASCADING-CUT and the last call of CASCADING-CUT may have a marked node)
    - ★ Change in potential is at most: $((t(H) + c) + 2(m(H) + c - 2)) - (t(H) + 2m(H)) = 4 - c$
    - ★ $\Rightarrow$ amortized cost of FIB-HEAP-DECREASE-KEY is $\mathcal{O}(c) + 4 - c = \mathcal{O}(1)$

SAMSUNG

### Lemma

*Let $x$ be any node in a Fibonacci heap, and suppose that degree$(x) = k$. Let $y_1, y_2, \ldots, y_k$ denote the children of $x$ in the order in which they were linked to $x$, from the earliest to the latest. Then, degree$(y_1) \geq 0$ and degree$(y_i) \geq i - 2, \forall i = 2, 3, \ldots, k$*

### Lemma

*$F_{k+2} = 1 + \sum_{i=0}^{k} F_i, \forall k \geq 0$ where $F_k$ is the $k^{th}$ element in the fibonacci sequence.*

### Lemma

*$F_{k+2} \geq \Phi^k$ where $\Phi = \frac{1+\sqrt{5}}{2}$ is a positive root of the equation $y^2 = y + 1$*

# Fibonacci heap - Bounding the maximum degree

SAMSUNG

## Lemma

*Let $x$ be any node in a fibonacci heap, and let $k = degree(x)$. Then $size(x) \geq F_{k+2} \geq \Phi^k$ where $\Phi = \frac{1+\sqrt{5}}{2}$*

## Corollary

*The maximum degree $D(n)$ of any node in a n-node fibonacci heap is $\mathcal{O}(lgn)$*

# Summary

| Procedures | Binary heap Worst case | Fibonacci heap Amortized |
|---|---|---|
| MAKE-HEAP | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |
| INSERT | $\mathcal{O}(logn)$ | $\mathcal{O}(1)$ |
| MINIMUM | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |
| EXTRACT-MIN | $\mathcal{O}(logn)$ | $\mathcal{O}(logn)$ |
| UNION | $\mathcal{O}(n)$ | $\mathcal{O}(1)$ |
| DECREASE-KEY | $\mathcal{O}(logn)$ | $\mathcal{O}(1)$ |
| DELETE | $\mathcal{O}(logn)$ | $\mathcal{O}(logn)$ |

# Disjoint sets

- Each set is represented by a rooted tree
  - Each node has a pointer to its parent
  - The root of each tree contains the representative and is its own parent
- Two heuristics to achieve asymptotically optimal disjoint-set data structure
  - union by rank
  - path compression

# Disjoint sets

- **Union by rank**
  - Maintain, for each node, a **rank** which is an upper bound on the height of the nodes
  - Make the root with smaller rank point to the root with larger rank during a UNION operation.
- **Path compression**
  - Used during the FIND-SET operation to make each node on the find path point directly to the root
  - Do not change any ranks

# Disjoint sets - implementation

- For each node $x$
  - $rank(x)$ is an upper bound on the height of $x$ (number of edges in the longest simple path from $x$ and its descendant leaf)
  - $p(x)$ is the parent of $x$

**Algorithm 15:** MAKE-SET($x$)

$p(x) \leftarrow x$;
$rank(x) = 0$;

**Algorithm 16:** FIND-SET($x$)

**if** $x \neq p(x)$ **then**
  $p(x) \leftarrow$ FIND-SET($p(x)$);
**return** $p(x)$;

# Disjoint sets - implementation

**Algorithm 17:** UNION$(x, y)$

LINK(FIND-SET$(x)$,FIND-SET$(y)$);

**Algorithm 18:** LINK$(x, y)$

**if** $rank(x) > rank(y)$ **then**
  $p(y) \leftarrow x$;
**else**
  $p(x) \leftarrow y$;
  **if** $rank(x) = rank(y)$ **then**
    $rank(y) \leftarrow rank(y) + 1$;

# Disjoint Set - implementation

```
package week12;
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
class IElement<T> {
  int rank;
  T parent;
  IElement(T parent, int rank) {
    this.parent = parent;
    this.rank = rank;
  }
}

public class DisjointSet<T> {
  private HashMap<T, IElement<T>> map = new HashMap<>();

  public void makeSet(T e) {
    map.put(e, new IElement<T>(e, 0));
  }
  . . .
}
```

# Disjoint Set - implementation

```
  public T find(T e) {
    IElement<T> ie = map.get(e);
    if (ie == null)
      return null;
    if (e != ie.parent)
      ie.parent = find(ie.parent);
    return ie.parent;
  }
  public void union(T x, T y) {
    if(x == y) return;
    T X = find(x);
    T Y = find(y);
    if (X == null || Y == null || X == Y) return;
    IElement<T> iX = map.get(X);
    IElement<T> iY = map.get(Y);
    if (iX.rank > iY.rank)
      iY.parent = x;
    else {
      iX.parent = y;
      if (iX.rank == iY.rank)
        iY.rank++;
    }
  }
}
```

# Dijkstra algorithm - implementation

```java
package week13;

import java.io.File;
import java.io.PrintWriter;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Scanner;

import week12.MinHeap;
import week12.Node;

public class Dijkstra {

  private HashSet<Node> V;
  private HashMap<Node, HashSet<Arc>> A;
  . . .
}
```

# Dijkstra algorithm - implementation

```
  public void findPath(Node s, Node t){
    MinHeap<Node> H = new MinHeap<Node>();
    s.key = 0;
    for(Arc a: A.get(s)){
      Node v = a.v;
      v.key = a.w;
      H.insert(v);
    }
    HashSet<Node> fixed = new HashSet<Node>();
    fixed.add(s);
    while(true){
      Node u = H.deleteMin();
      fixed.add(u);
      if(u == t) break;
      for(Arc a: A.get(u)){
        if(!fixed.contains(a.v) && a.v.key > u.key + a.w){
            a.v.key = u.key + a.w;
            if(!H.contains(a.v)) H.insert(a.v);
            else H.decreaseKey(a.v);
        }
      }
    }
    System.out.println("Shortest distance = " + t.key);
  }
}
```

# Kruskal algorithm - implementation

```java
package week13;

import java.io.File;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Scanner;
import java.util.HashSet;

import week12.DisjointSet;
import week12.MinHeap;
import week12.Node;

public class Kruskal {

  HashSet<Node> V;
  Edge[] E;

  . . .
}
```

# Kruskal algorithm - implementation

```
  public void findMST(){
    MinHeap H = new MinHeap();
    H.sort(E);
    DisjointSet<Node> DS = new DisjointSet<Node>();
    for(Node v: V)
      DS.makeSet(v);

    int W = 0;
    HashSet<Edge> T = new HashSet<Edge>();
    for(int i = 0;i < E.length; i++){
      if(DS.find(E[i].u) == DS.find(E[i].v)) continue;
      T.add(E[i]);
      W += E[i].w;
      DS.union(E[i].u, E[i].v);
      if(T.size() == V.size() - 1) break;
    }
    System.out.println("W = " + W);
  }
}
```

# Prim algorithm - implementation

```
package week13;

import java.io.File;
import java.io.PrintWriter;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Scanner;

import week12.MinHeap;
import week12.Node;

public class Prim {

  private HashSet<Node> V;
  private HashMap<Node, HashSet<Arc>> A;
  . . .
}
```

# Prim algorithm - implementation

```
  public void findMST(Node s, Node t){
    MinHeap<Node> H = new MinHeap<Node>();
    s.key = 0;
    for(Arc a: A.get(s)){
      Node v = a.v;
      v.key = a.w;
      H.insert(v);
    }
    HashSet<Node> fixed = new HashSet<Node>();
    fixed.add(s);
    int W = 0;
    while(true){
      Node u = H.deleteMin();
      W += u.key;
      fixed.add(u);
      if(u == t) break;
      for(Arc a: A.get(u)){
        if(!fixed.contains(a.v) && a.v.key > a.w){
            a.v.key = a.w;
            if(!H.contains(a.v)) H.insert(a.v);
            else H.decreaseKey(a.v);
        }
      }
    }
  }
}
```