Cấu trúc dữ liệu nâng cao và Ứng dụng

Khoa Công Nghệ Thông Tin,
Trường Đại Học Thủy Lợi.

Ngày 9 tháng 3 năm 2017

Nội dung

SAMSUNG

- Hàng đợi ưu tiên
- Tập rời rạc
- Ứng dụng
  - Thuật toán Dijkstra
  - Thuật toán Kruskal
  - Thuật toán Prim

Hàng đợi ưu tiên

- Cấu trúc dữ liệu lưu trữ tập các phần tử mà mỗi phần tử được liên kết với một khóa và các thao tác sau:
  - Chèn một phần tử
  - Trả về một phần tử với khóa nhỏ nhất
  - Trả về một phần tử với khóa nhỏ nhất và loại bỏ phần tử này ra khỏi hàng đợi
  - Giảm khóa của một phần tử
- Ứng dụng
  - Thuật toán Dijkstra để tìm những đường đi ngắn nhất trong một đồ thị
  - Thuật toán Prim để tìm cây bao trùm nhỏ nhất của một đồ thị
  - Sắp xếp đống
  - ...

Đống nhị phân

- Cây nhị phân hoàn thiện
  - Tất cả cấp độ, có thể trừ phần tử cuối cùng, được điền đầy đủ
  - Nếu cấp độ cuối cùng không được điền đầy đủ, các nút của cấp độ đó được điền từ trái qua phải
- Thuộc tính đống
  - Khóa của mỗi nút nhỏ hơn hoặc bằng khóa của những nút con của nó
  - Phần tử nhỏ nhất là gốc
  - Đống (Heap) với n phần tử có chiều cao $\lfloor logn \rfloor$

Đống nhị phân - Cài đặt

- Sử dụng một mảng $x[1..n]$
  - $parent(x[i]) = x[(i)/2]$
  - $leftChild(x[i]) = x[2i]$
  - $rightChild(x[i]) = x[2i+1]$
- Chèn, Giảm-Khóa, Lấy-ra-phần-tử-nhỏ-nhất: $\mathcal{O}(logn)$
- Tìm-phần-tử-nhỏ-nhất: $\mathcal{O}(1)$

Đống nhị phân - Cài đặt

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

```
t ← x[k];
while k < n do
    c ← 2 × k;
    if c < n ∧ x[c + 1] < x[c] then
        c ← c + 1;
    if c < n ∧ t > x[c] then
        x[k] ← x[c];
    else
        BREAK;
    k ← c;
x[k] ← t;
```

Đống nhị phân - Cài đặt

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

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

Đống nhị phân - Cài đặt

SAMSUNG

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;

Đống nhị phân - Cài đặt

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;
```

Đống nhị phân - Cài đặt

```
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 no
  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();
  }
  . . .
}
```

Đống nhị phân - Cài đặt

```
  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);
  }
  . . .
}
```

Đống nhị phân - Cài đặt

```
  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);
  }
  . . .
}
```

Đống nhị phân - Cài đặt

```
  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);
  }
  . . .
}
```

Đống nhị phân - Cài đặt

```
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;
}
. . .
}
```

Đống nhị phân - Cài đặt

```
  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);
  }
}
```

Đống Fibonacci

- Tập hợp những cây có gốc **trật tự đống-nhỏ-nhất**
- Với mỗi nút x
  - $p(x)$: cha của x
  - $child(x)$: chỉ tới một trong các nút con của x
  - Những nút con của x được liên kết với nhau trong một chu trình (danhh sách liên kết kép, được gọi là danh sách nút con của x)
  - $left(x)$ và $right(x)$: chỉ tới những nút anh/chị trái và phải của x
  - $degree(x)$: số lượng nút con của x
  - $mark(x)$: TRUE nếu x mất một nút con do lần cuối cùng x được tạo nút con của những nút khác
    - ★ Một nút được tạo mới y là không được đánh dấu: $mark(y) = \text{FALSE}$
    - ★ Một nút y trở thành không được đánh dấu bất cứ khi nào nó được tạo nút con của nút khác
    - ★ Thuộc tính $mark(.)$ là tập FALSE khi chúng ta xét những thao tác GIẢM-KHÓA

Đống Fibonacci

- Với mỗi đống fibonacci H
  - Gốc của tất cả các cây được liên kết với nhau trong một danh sách liên kết kép, được gọi là **danh sách gốc** của đống fibonacci
  - Các cây có thể xuất hiện theo bất cứ trật tự nào trong danh sách gốc
  - $min(H)$ là một con trỏ trỏ tới gốc của một cây chứa khóa nhỏ nhất (được gọi là **nút nhỏ nhất** của đống fibonacci)
  - $n(H)$: số lượng nút của H
  - $t(H)$: số lượng cây trong danh sách gốc của H
  - $m(H)$: số lượng nút được đánh dấu của H
  - Hàm khả năng: $\Phi(H) = t(H) + 2m(H)$
- Chúng ta biểu thị $D(n)$ là bậc lớn nhất của bất kỳ nút nào trong một đống fibonacci n-nút
- Phương trình sau sẽ được chứng minh $D(n) = \mathcal{O}(lgn)$

Đống Fibonacci - Cài đặt các thao tác

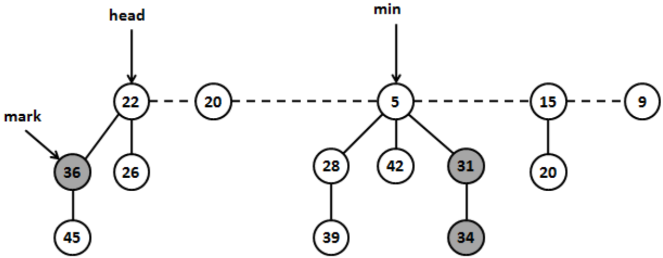
- Tạo một đống rỗng
- $\mathcal{O}(1)$

Algorithm 5: FIB-HEAP-MAKE()

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

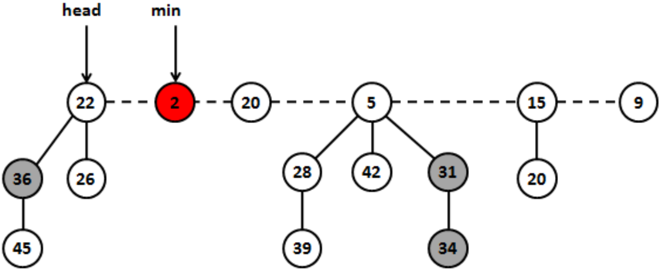
Đống Fibonacci - Cài đặt các thao tác

Đống hiện tại



Đống Fibonacci - Cài đặt các thao tác

Chèn nút với khóa $= 2$



Đống Fibonacci - Cài đặt các thao tác

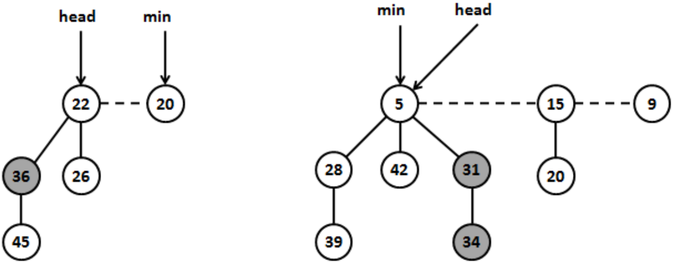
- Chèn một nút x vào đống H
- Không hợp nhất như trong heap nhị thức (lazzy)
- Phân tích (H' là heap kết quả)
  - Tăng khả năng là $t(H') + 2m(H') - (t(H) + 2m(H)) = t(H) + 1 + 2m(H) - (t(H) + 2m(H)) = 1$
  - $\Rightarrow$ Giá trị khấu hao nếu $\mathcal{O}(1) + 1 = \mathcal{O}(1)$ (bởi vì giá trị thực tế là $\mathcal{O}(1)$)

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**
    Tạo một danh sách gốc cho H chỉ chứa x;
    $min(H) \leftarrow x$;
else
    Chèn x vào trong danh sách gốc của H;
    if $key(x) < key(min(H))$ **then**
        $min(H) \leftarrow x$;
    $n(H) \leftarrow n(H) + 1$;

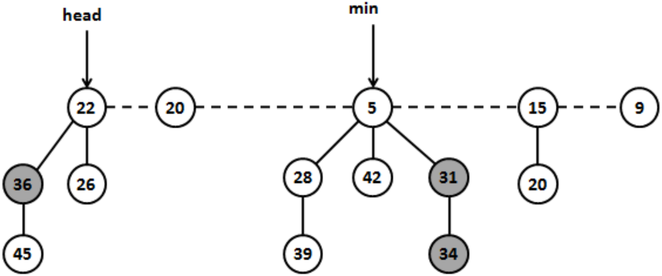
Đống Fibonacci - Cài đặt các thao tác

Hai đống fibonacci được hợp nhất



Đống Fibonacci - Cài đặt các thao tác

Sau khi kết hợp



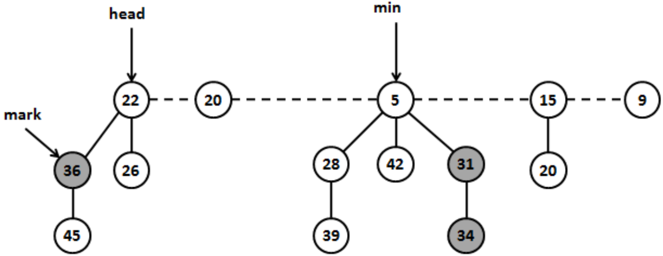
Đống Fibonacci - Cài đặt các thao tác

- Kết hợp hai đống
- Phân tích
  - Thay đổi khả năng là $\Phi(H) - \Phi(H_1) - \Phi(H_2) = 0$
  - Do giá trị thực tế là $\mathcal{O}(1) \Rightarrow$ giá trị khấu hao là $\mathcal{O}(1)$

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

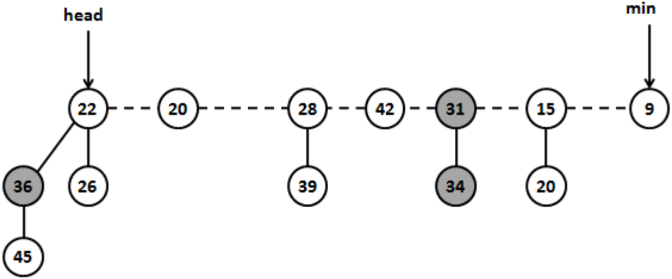
$H \leftarrow$ FIB-HEAP-MAKE();
$min(H) \leftarrow min(H_1)$;
Nối danh sách gốc của H_2 với danh sách gốc của H_1;
if $min(H_1) = NIL$ *hoặc* $min(H_2) \neq NIL$ *và* $key(min(H_2)) < key(min(H_1))$
then
    $min(H) \leftarrow min(H_2)$;
$n(H) \leftarrow n(H_1) + n(H_2)$;
return H;

ExtractMin



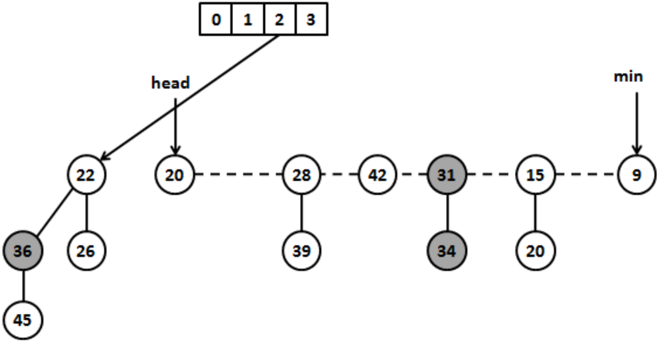
Đống Fibonacci - Cài đặt các thao tác

ExtractMin: Củng cố - bước 1



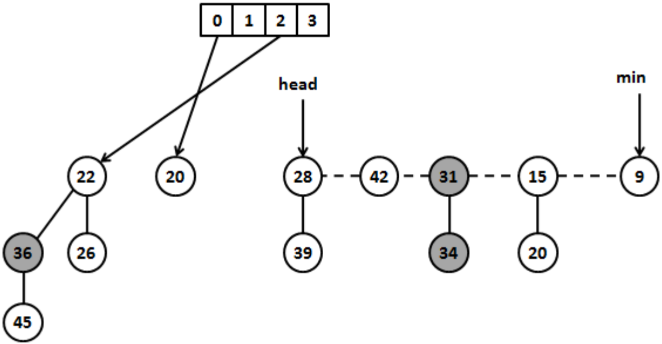
Đống Fibonacci - Cài đặt các thao tác

ExtractMin: Củng cố - bước 2



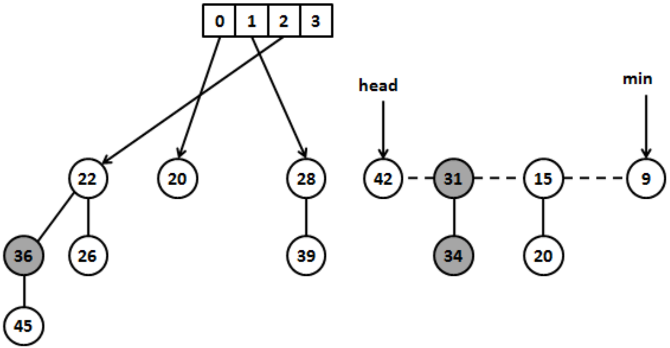
Đống Fibonacci - Cài đặt các thao tác

ExtractMin: Củng cố - bước 3



Đống Fibonacci - Cài đặt các thao tác

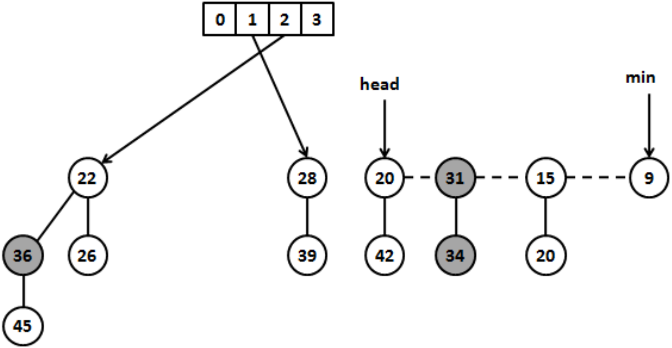
ExtractMin: Củng cố - bước 4



Đống Fibonacci - Cài đặt các thao tác

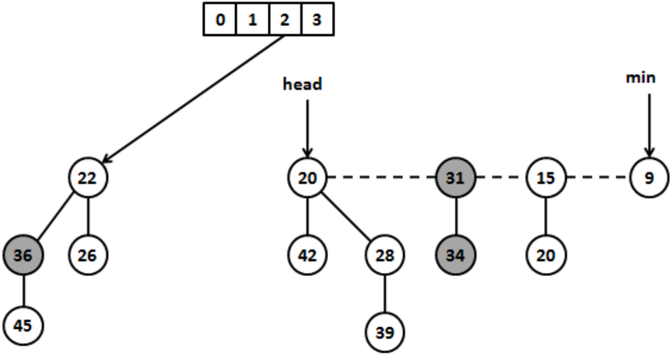
SAMSUNG

ExtractMin: Củng cố - bước 5



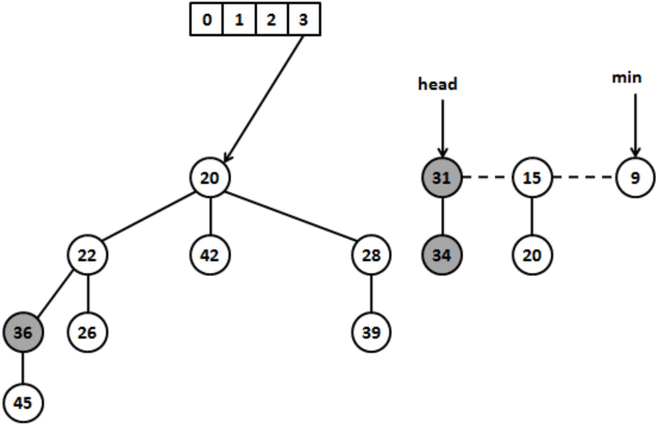
Đống Fibonacci - Cài đặt các thao tác

ExtractMin: Củng cố - bước 6



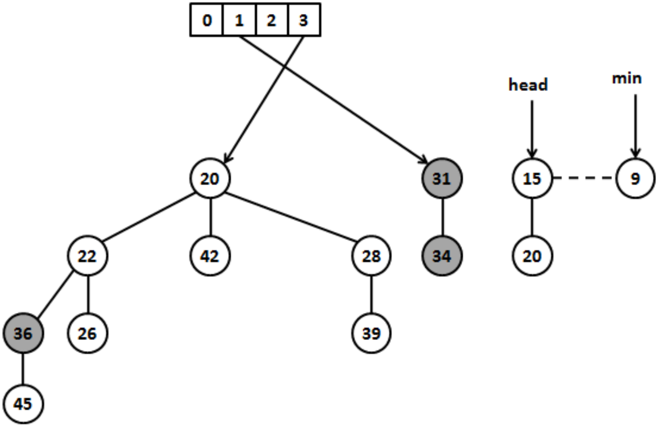
Đống Fibonacci - Cài đặt các thao tác

ExtractMin: Củng cố - bước 7



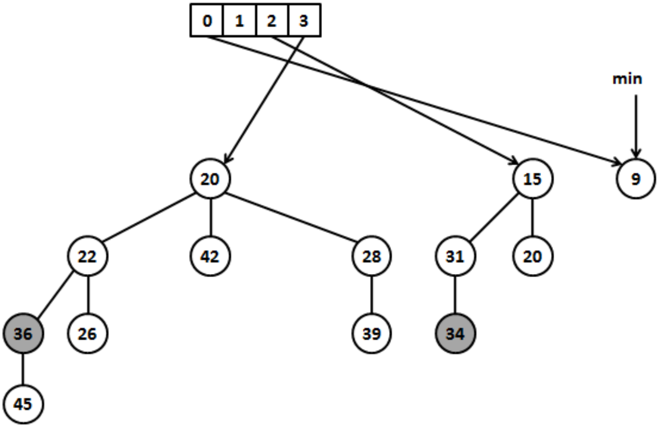
Đống Fibonacci - Cài đặt các thao tác

ExtractMin: Củng cố - bước 8



Đống Fibonacci - Cài đặt các thao tác

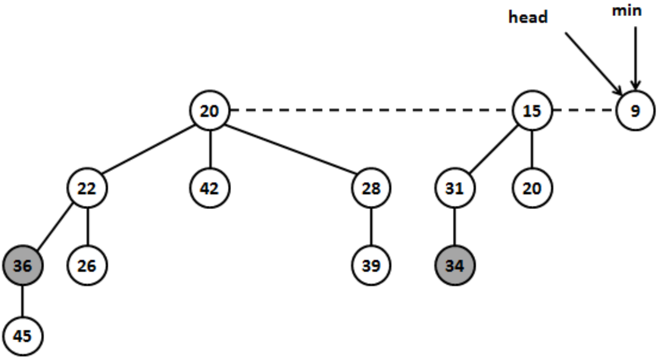
ExtractMin: Củng cố - bước 9



Đống Fibonacci - Cài đặt các thao tác

SAMSUNG

ExtractMin: Củng cố - bước 10



Đống Fibonacci - Cài đặt các thao tác

- Trích xuất phần tử nhỏ nhất (thao tác phức tạp nhất)
- Công việc bị trì hoãn của việc hợp nhất cây trong danh sách gốc cuối cùng cũng xảy ra
  - Liên kết những gốc có bậc bằng nhau cho đến khi có tối đa một gốc cho một bậc

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

$z \leftarrow min(H)$;
if $z \neq NIL$ **then**
  foreach *con* x *của* z **do**
    Thêm x vào trong danh sách gốc của H;
    $p(x) \leftarrow$ NIL;
  Loại bỏ z từ danh sách gốc của H;
  if $z = right(z)$ **then**
    $min(H) \leftarrow$ NIL;
  else
    $min(H) \leftarrow right(z)$;
    CONSOLIDATE(H);
  $n(H) \leftarrow n(H) - 1$;

Đống Fibonacci - Cài đặt các thao tác

Algorithm 9: CONSOLIDATE(H)

Cho $A[0..D(n(H))]$ là một mảng mới;
foreach $i \in \{0, \ldots, D(n(H))\}$ **do**
    $A[i] \leftarrow$ NIL;

foreach *nút w trong danh sách gốc của H* **do**
    $x \leftarrow w$;
    $d \leftarrow degree(x)$;
    while $A[d] \neq NIL$ **do**
        $y \leftarrow A[d]$;
        if $key(x) > key(y)$ **then**
            hoán đổi x với y;
        FIB-HEAP-LINK(H, y, x);
        $A[d] \leftarrow$ NIL;
        $d \leftarrow d + 1$;
    $A[d] \leftarrow x$;
$min(H) \leftarrow$ NIL;
foreach $i \in \{0, \ldots, D(n(H))\}$ **do**
    if $A[i] \neq NIL$ **then**
        if $min(H) = NIL$ **then**
            Tạo một danh sách gốc cho H chỉ chứa $A[i]$;
            $min(H) \leftarrow A[i]$;
        else
            Chèn $A[i]$ vào trong danh sách gốc của H;
            if $key(A[i]) < key(min(H))$ **then**
                $min(H) \leftarrow A[i]$;

Đống Fibonacci - Cài đặt các thao tác

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

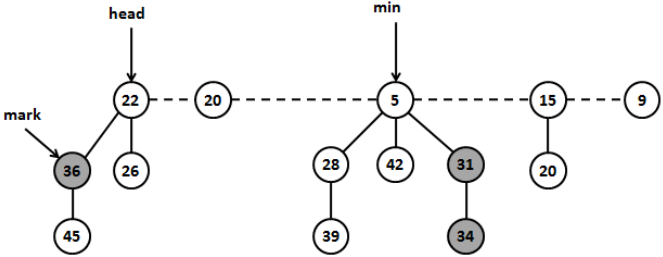
Loại bỏ y từ danh sách gốc của H;
Tạo y là con của x;
$degree(x) \leftarrow degree(x) + 1$;
$mark(y) \leftarrow$ FALSE;

Đống Fibonacci - Cài đặt các thao tác

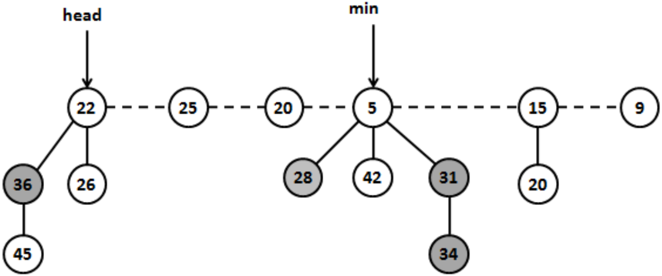
- Phân tích hàm CONSOLIDATE
  - Kích thước của danh sách gốc khi gọi hàm CONSOLIDATE tối đa là $\mathcal{O}(D(n)) + t(H) - 1$, do nó gồm $t(H)$ nút của danh sách gốc, trừ đi nút được trích xuất, cộng nút con của nút được trích xuất đó là $\mathcal{O}(D(n))$
  - vòng lặp **for** từ dòng 4 đến dòng 14
    - ★ Mỗi lần chạy qua vòng lặp **while** từ dòng 7 đến dòng 13, một trong các gốc được liên kết với gốc khác
    - ★ $\Rightarrow$ Tổng lượng công việc là tỷ lệ với $D(n) + t(H)$
  - vòng lặp **for** từ dòng 6 đến dòng 24: $\mathcal{O}(D(n))$
- $\Rightarrow$ EXTRACT-MIN thực sự tốn $\mathcal{O}(D(n) + t(H))$
- Khả năng ở trước EXTRACT-MIN là $t(H) + 2m(H)$ và ở sau là tối đa $D(n) + 1 + 2m(H)$, do tối đa $D(n) + 1$ gốc còn lại và không nút nào trở thành được đánh dấu trong suốt thao tác
- Giá trị khấu hao là $\mathcal{O}(D(n) + t(H)) + (D(n) + 1 + 2m(H)) - (t(H) + 2m(H)) = \mathcal{O}(D(n))$

Đống Fibonacci - Cài đặt các thao tác

Đống hiện tại

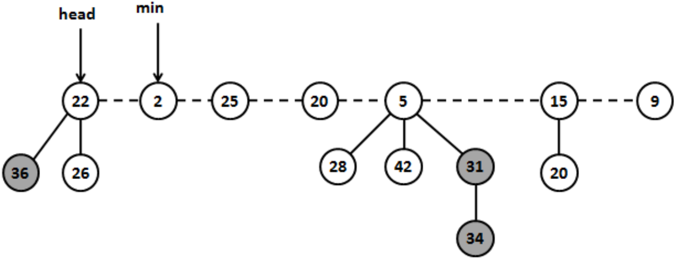


DecreaseKey của 39 thành 25



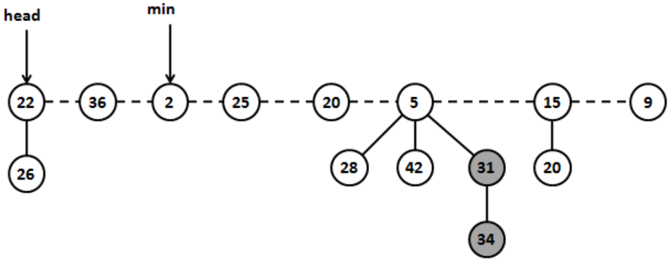
Đống Fibonacci - Cài đặt các thao tác

DecreaseKey của 45 thành 2



Đống Fibonacci - Cài đặt các thao tác

DecreaseKey của 45 thành 2



Đống Fibonacci - Cài đặt các thao tác

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

```
if key(x) < k then
    error;
key(x) ← k;
y ← p(x);
if y ≠ NIL và key(x) < key(y) then
    CUT(H, x, y);
    CASCADING-CUT(H, y);
if key(x) < key(min(H)) then
    min(H) ← x;
```

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

```
FIB-HEAP-DECREASE-KEY(H, x, −∞);
FIB-HEAP-EXTRACT-MIN(H);
```

Đống Fibonacci - Cài đặt các thao tác

Algorithm 13: CUT(H, x, y)

Loại bỏ x từ danh sách con của y;
$degree(y) \leftarrow degree(y) - 1$;
Thêm x vào danh sách gốc của H;
$p(x) \leftarrow$ NIL;
$mark(x) \leftarrow$ FALSE;

Algorithm 14: CASCADING-CUT(H, y)

$z \leftarrow p(y)$;
if $z \neq NIL$ **then**
    if $mark(y) = FALSE$ **then**
        $mark(y) \leftarrow$ TRUE;
    else
        CUT(H, y, z);
        CASCADING-CUT(H, z);

Đống Fibonacci - Cài đặt các thao tác

- Phân tích hàm FIB-HEAP-DECREASE-KEY
  - Giả sử rằng hàm CASCADING-CUT được gọi đệ quy c lần từ một lời gọi cho trước của hàm FIB-HEAP-DECREASE-KEY
  - Mỗi lời gọi hàm CASCADING-CUT tốn $\mathcal{O}(1)$ không bao gồm lời gọi đệ quy
  - Do đó, giá trị thực tế của hàm FIB-HEAP-DECREASE-KEY là $\mathcal{O}(c)$
  - Thay đổi khả năng
    - ★ Mỗi lời gọi đệ quy của hàm CASCADING-CUT, ngoại trừ lần cuối cùng, cắt một nút được đánh dấu và bỏ bit đánh dấu
    - ★ Sau đó, có $t(H) + c$ cây ($t(H)$ cây ban đầu, $c - 1$ cây được tạo ra bởi hàm CASCADING-CUT và cây có gốc tại x) và tối đa $m(H) + c - 2$ nút được đánh dấu ($c - 1$ được bỏ đánh dấu bởi CASCADING-CUT và lời gọi cuối cùng của CASCADING-CUT có thể có một nút đánh dấu)
    - ★ Thay đổi khả năng tối đa là: $((t(H) + c) + 2(m(H) + c - 2)) - (t(H) + 2m(H)) = 4 - c$
    - ★ $\Rightarrow$ giá trị khấu hao của hàm FIB-HEAP-DECREASE-KEY là $\mathcal{O}(c) + 4 - c = \mathcal{O}(1)$

Đống Fibonacci - Giới hạn bậc lớn nhất

SAMSUNG

Lemma

Đặt x là bất kỳ nút nào trong một đống Fibonacci, và giả sử rằng $degree(x) = k$. *Đặt* $y_1, y_2, \ldots, y_k$ *biểu thị những nút con của x theo một trật tự mà chúng được liên kết với x, từ lúc sớm nhất đến muộn nhất. Do đó,* $degree(y_1) \geq 0$ *và* $degree(y_i) \geq i - 2, \forall i = 2, 3, \ldots, k$

Lemma

$F_{k+2} = 1 + \sum_{i=0}^{k} F_i, \forall k \geq 0$ *với* F_k *là phần tử thứ k trong chuỗi fibonacci.*

Lemma

$F_{k+2} \geq \Phi^k$ *với* $\Phi = \frac{1+\sqrt{5}}{2}$ *là một gốc dương của phương trình* $y^2 = y + 1$

Đống Fibonacci - Giới hạn bậc lớn nhất

Lemma

Đặt x là một nút bất kỳ trong một đống fibonacci, và đặt $k = degree(x)$. Sau đó $size(x) \geq F_{k+2} \geq \Phi^k$ với $\Phi = \frac{1+\sqrt{5}}{2}$

Corollary

Bậc lớn nhất $D(n)$ của bất kỳ nút nào trong một đống fibonacci n-nút là $\mathcal{O}(lgn)$

Tóm tắt

Hàm	Heap nhị phân Trường hợp xấu nhất	Heap Fibonacci Khấu hao
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	$\mathcal{O}(logn)$	$\mathcal{O}(logn)$

Tập rời rạc

- Mỗi tập được biểu diễn bởi một cây gốc (rooted tree)
  - Mỗi nút có một con trỏ tới nút cha của nó
  - Gốc của mỗi cây chứa đại diện và là nút cha của chính nó
- Hai heuristic để đạt được một cách tiệm cận cấu trúc dữ liệu tập-rời-rạc tối ưu
  - hợp nhất bởi thứ bậc
  - nén đường đi

- **Hợp nhất bởi thứ bậc**
  - Duy trì, với mỗi nút, một **thứ bậc** là giới hạn trên về chiều cao của các nút
  - Tạo gốc với thứ bậc nhỏ hơn trỏ đến gốc với thứ bậc lớn hơn trong một thao tác HỢP NHẤT.
- **Nén đường đi**
  - Được sử dụng trong thao tác FIND-SET để tạo mỗi nút trên đường đi tìm kiếm trỏ trực tiếp tới gốc
  - Không thay đổi bất kỳ thứ bậc nào

Tập rời rạc - Cài đặt

- Với mỗi nút x
  - $rank(x)$ là một giới hạn trên về chiều cao của x (số lượng cạnh trong đường đi đơn giản dài nhất từ x và nút lá hậu duệ của nó)
  - $p(x)$ là nút cha của 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 $p(x)$;

Tập rời rạc - Cài đặt

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**
    $rank(y) \leftarrow rank(y) + 1$;

Tập rời rạc - Cài đặt

```java
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));
  }
  . . .
}
```

Tập rời rạc - Cài đặt

```
  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++;
    }
  }
}
```

Thuật toán Dijkstra - Cài đặt

```
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;
  . . .
}
```

Thuật toán Dijkstra - Cài đặt

```
  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);
  }
}
```

Thuật toán Kruskal - Cài đặt

```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;

  . . .
}
```

Thuật toán Kruskal - Cài đặt

```
  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);
  }
}
```

Thuật toán Prim - Cài đặt

```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 Prim {

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

Thuật toán Prim - Cài đặt

```
  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);
        }
      }
    }
  }
}
```