ĐẠI HỌC QUỐC GIA TP.HCM
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN

——————o0o——————

Course Project - Data Structures and Algorithms - CSC10004

Priority Queue Using Min/Max Binary Heap

Member:	Phan Ngưng	ID: 23120304
	Võ Quốc Gia	ID: 23120014
	Trần Minh Hiếu Học	ID: 23120006
Class:	CNTN2023	

TP Hồ Chí Minh, 12/2024

Contents

1 Mở đầu

Trong lĩnh vực khoa học máy tính và lập trình, cấu trúc dữ liệu đóng vai trò quan trọng trong việc tối ưu hóa hiệu suất của các thuật toán và ứng dụng. **Priority Queue** (Hàng đợi ưu tiên) là một trong những cấu trúc dữ liệu phổ biến, được sử dụng rộng rãi trong các ứng dụng yêu cầu quản lý các phần tử theo thứ tự ưu tiên, chẳng hạn như lập lịch CPU, thuật toán đường đi ngắn nhất (Dijkstra), hoặc các bài toán tổ hợp.

Trong bài báo cáo này, nhóm chúng mình sẽ tập trung nghiên cứu **Binary Heap**, một cấu trúc dữ liệu đặc biệt để triển khai Priority Queue hiệu quả. Binary Heap cung cấp cách tiếp cận nhanh chóng và tối ưu để thực hiện các thao tác cơ bản như chèn phần tử (**insert**) và lấy phần tử có độ ưu tiên cao nhất (**extract-min/max**). Cấu trúc này không chỉ đảm bảo tính chính xác về mặt logic mà còn mang lại hiệu quả vượt trội về mặt thời gian xử lý.

Bài báo cáo sẽ trình bày chi tiết về nguyên lý hoạt động, các thao tác chính, và ứng dụng thực tiễn của **Priority Queue** dựa trên **Binary Heap**. Đồng thời, các ví dụ minh họa và phân tích phức tạp thời gian sẽ giúp làm rõ hơn ưu điểm của cấu trúc dữ liệu này trong các bài toán thực tế.

2 Lịch sử hình thành của Priority Queue

John William Joseph Williams, thường được gọi là J. W. J. Williams, là một nhà khoa học máy tính người Anh, sinh tháng 9 năm 1930 và qua đời ngày 29 tháng 9 năm 2012. Ông là một trong những nhà tiên phong trong lĩnh vực thuật toán và cấu trúc dữ liệu, đặc biệt nổi bật với đóng góp quan trọng liên quan đến Heap Sort và Binary Heap.

Năm 1964, khi đang công tác tại Elliot Brothers Ltd ở London, Williams công bố thuật toán Heap Sort trên tạp chí Communications of the ACM với tiêu đề **"Algorithm 232: Heapsort"**. Đây là một bước đột phá lớn trong lĩnh vực khoa học máy tính, giúp cải tiến hiệu quả sắp xếp dữ liệu. Heap Sort là thuật toán đầu tiên áp dụng cấu trúc Binary Heap (đống nhị phân) để sắp xếp dữ liệu. Binary Heap không chỉ hỗ trợ sắp xếp nhanh chóng mà còn trở thành nền tảng cho việc triển khai Priority Queue (hàng đợi ưu tiên).

2.1 Mối liên hệ Priority Queue và Binary Heap

Cấu trúc binary heap do Williams giới thiệu có hai tính chất nổi bật:

- Truy cập nhanh chóng phần tử quan trọng nhất: Binary Heap cho phép truy cập phần tử lớn nhất (trong max-heap) hoặc nhỏ nhất (trong min-heap) trong thời gian hằng số $O(1)$.
- Thao tác hiệu quả: Các thao tác chèn (insert) hoặc xóa (delete) đều được thực hiện trong thời gian $O(log\, n)$ nhờ vào tính chất phân cấp của heap.

Những đặc điểm này đã biến binary heap thành cấu trúc lý tưởng để triển khai hàng đợi ưu tiên. Hàng đợi ưu tiên là một trong những công cụ cốt lõi trong khoa học máy tính, được sử dụng để quản lý các công việc theo mức độ ưu tiên. Ví dụ, trong một hệ điều hành, hàng đợi ưu tiên có thể đảm bảo rằng các tác vụ quan trọng nhất sẽ được xử lý trước tiên.

2.2 Ý nghĩa từ công trình nghiên cứu của Williams

Thuật toán HeapSort và cấu trúc Binary Heap của Williams không chỉ mở ra các ứng dụng trong sắp xếp dữ liệu mà còn được ứng dụng rộng rãi trong:

- Hệ thống lập lịch CPU: Quản lý tác vụ theo mức độ ưu tiên.
- Thuật toán tìm đường ngắn nhất: Như thuật toán DIJKSTRA, sử dụng hàng đợi ưu tiên để tìm đường hiệu quả.
- Hệ điều hành và cơ sở dữ liệu: Quản lý tài nguyên hoặc xử lý các hàng đợi công việc.

Công trình của J. W. J. Williams không chỉ ảnh hưởng trực tiếp đến việc tối ưu hóa dữ liệu mà còn truyền cảm hứng cho các nghiên cứu về thuật toán và cấu trúc dữ liệu tiên tiến sau này. Binary heap đã trở thành nền tảng trong khoa học máy tính hiện đại, là biểu tượng cho sự kết hợp giữa tính sáng tạo và ứng dụng thực tế trong thiết kế thuật toán.

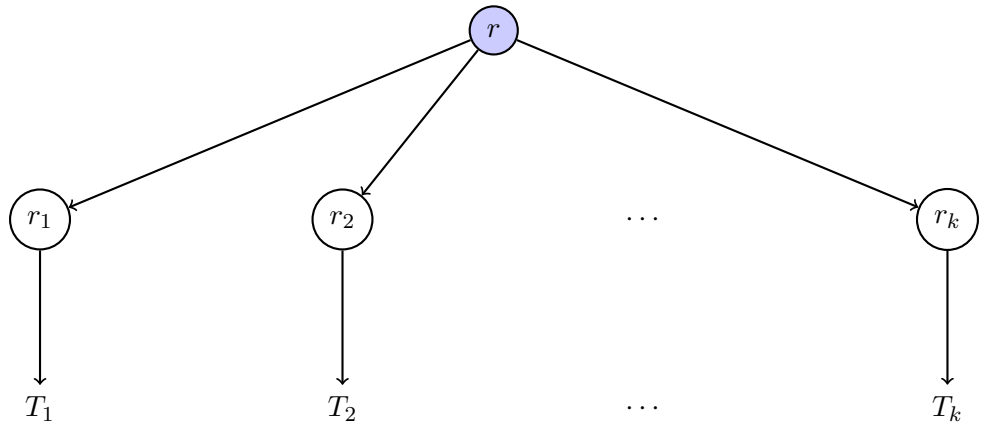
3 Một số định nghĩa - khái niệm

3.1 Cây

Định nghĩa

Cây bao gồm các nút, có một nút đặc biệt được gọi là nút gốc (root) và các cạnh nối các nút. Cây được định nghĩa đệ quy như sau:

- Bước cơ sở: một nút r được gọi là cây và r được gọi là gốc cây.
- Bước đệ quy: Giả sử $T_1, T_2, \ldots, T_k$ là các cây với gốc là $r_1, r_2, \ldots, r_k$. Ta có thể xây dựng cây mới bằng cách đặt r làm nút cha (parent) của các nút gốc $r_1, r_2, \ldots, r_k$. Trong cây mới tạo ra r là gốc và $T_1, T_2, \ldots, T_k$ là các cây con của gốc r. Các nút $r_1, r_2, \ldots, r_k$ được gọi là con của nút r.



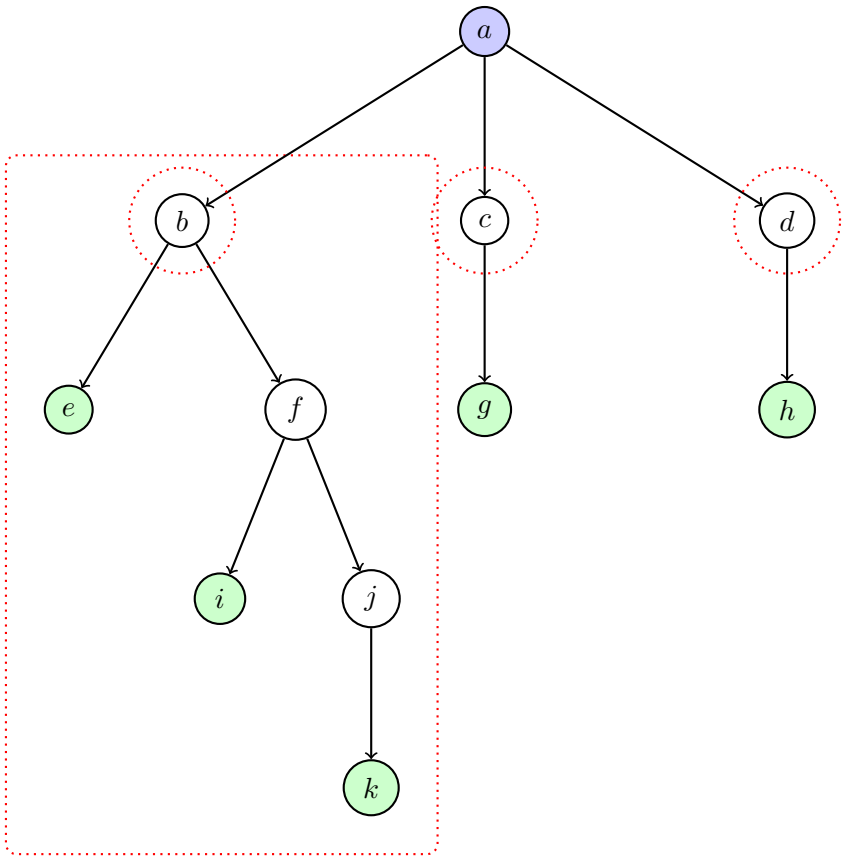
Hình minh họa định nghĩa đệ quy của cây

Các thuật ngữ

- Gốc (root): Nút đầu tiên tạo nên cây.
- Tổ tiên (ancestors): Nút A là tổ tiên của nút B nếu đường đi từ nút gốc đến B phải đi qua A.
- Cha (parent): Nút A là cha của nút B khi đường đi từ nút gốc đến B phải đi qua A và A được nối với B.
- Con (child): Nút có một nút cha.
- Lá (leaf): Nút không có nút con.
- Hậu duệ (descendants): B là hậu duệ của A khi A là tổ tiên của B.

- Anh em (sibling): Các nút có chung cha.
- Mức (level): Số cạnh từ gốc đến một nút.
- Chiều cao (height): Số cạnh từ gốc đến nút xa nhất.
- Nút trong (internal node): Là các nút không phải là nút lá.

Một vài ví dụ để giải thích thêm cho các thuật ngữ trên:



Nút xanh thẫm (a) là nút gốc, các nút xanh lá là nút lá, cây này có chiều cao là 4.

Các nút b, c và d là các nút anh em với nút cha là a.

Nhánh cây từ đỉnh b trở xuống là cây con của a có gốc b.

3.2 Cây nhị phân

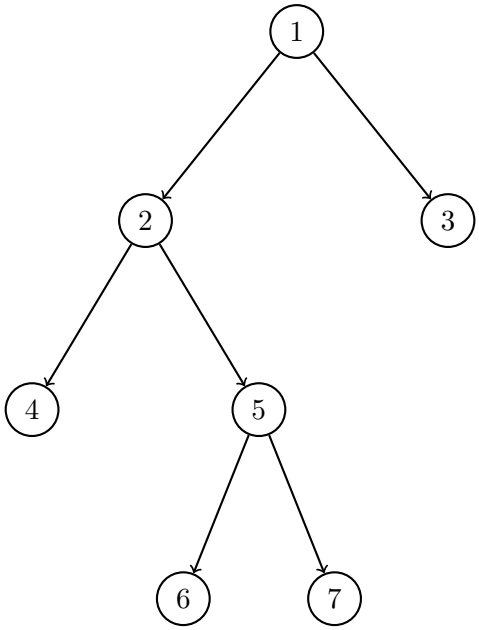
Định nghĩa

Cây nhị phân là một cấu trúc dữ liệu mà mỗi nút cha có tối đa 2 nút con.

Các thuật ngữ về cây nhị phân giống như các thuật ngữ về cây đã giới thiệu ở trên.

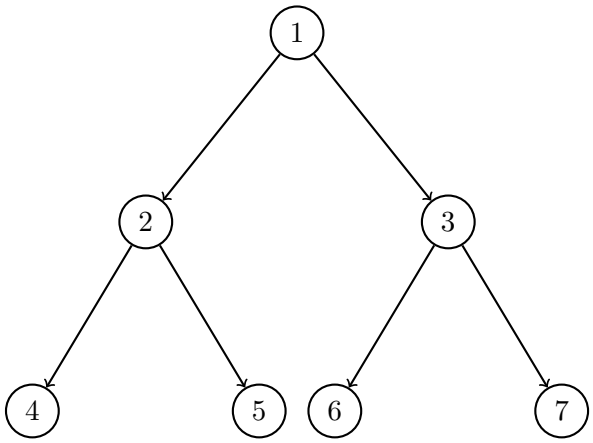
Các kiểu cây nhị phân

- Cây nhị phân đầy đủ (Full binary tree): Mỗi nút cha có 0 hoặc 2 nút con.



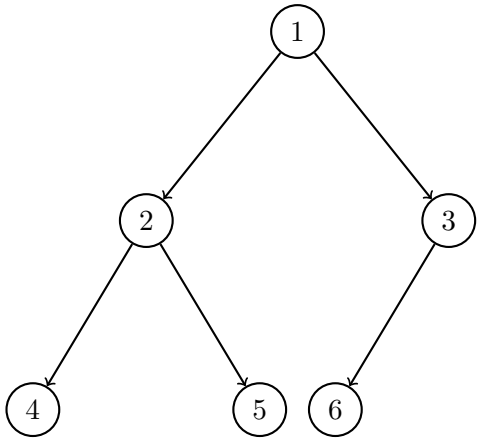
Ví dụ về cây nhị phân đầy đủ.

- Cây nhị phân hoàn hảo (Perfect binary tree): Mỗi nút trung gian đều có 2 con và mọi nút lá đều cùng mức.



Ví dụ về cây nhị phân hoàn hảo.

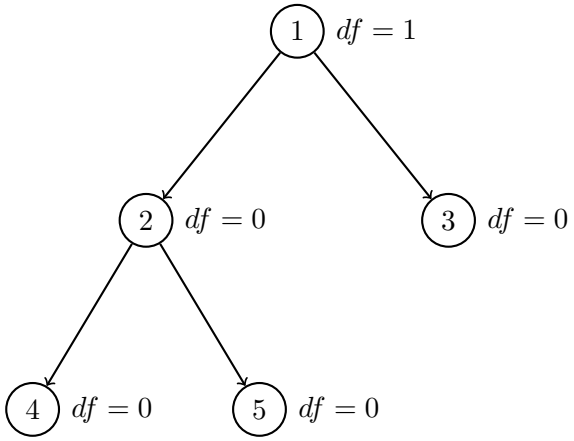
- Cây nhị phân hoàn chỉnh (Complete binary tree): Tất cả mức - trừ mức cuối - tạo thành một cây nhị phân hoàn hảo, với mức cuối các nút lá phải tinh gọn về bên trái.



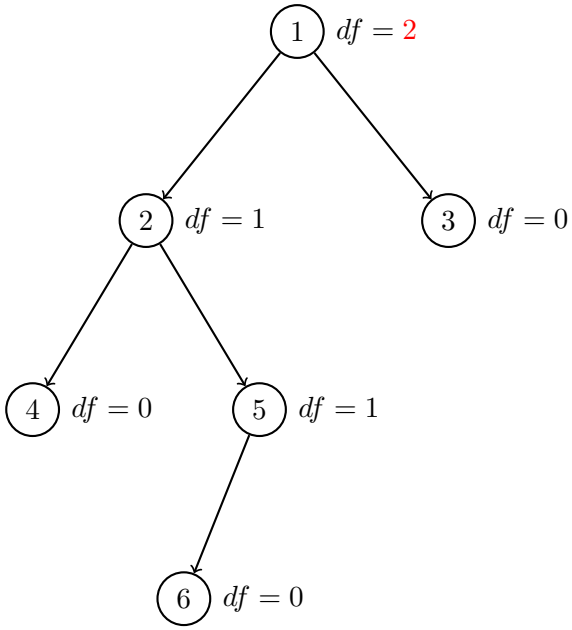
Ví dụ về cây nhị phân hoàn chỉnh.

- Cây nhị phân cân bằng (Balanced binary tree): Mọi nút trên cây có độ cao của cây con bên trái và độ cao của cây con bên phải chênh lệch nhau không quá 1.

$$df = (height\ of\ left\ child) - (height\ of\ right\ child)$$



Ví dụ về cây nhị phân cân bằng.



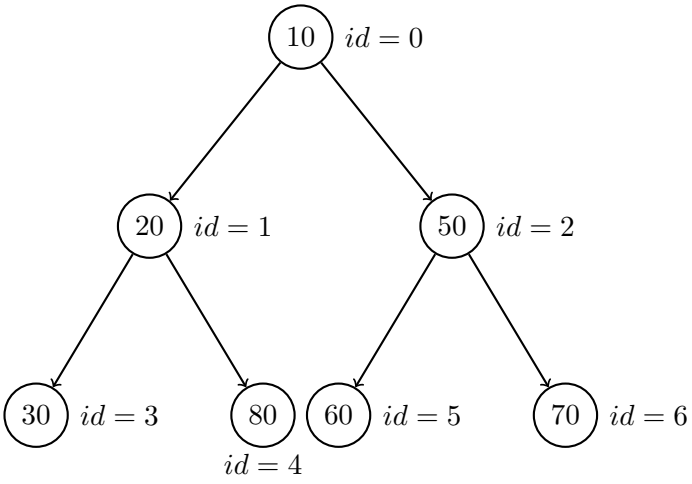
Ví dụ về cây nhị phân không cân bằng.

Trong bài báo cáo này, chủ yếu chỉ làm việc với Cây nhị phân và cụ thể hơn là Cây nhị phân hoàn chỉnh (Complete binary tree) để xây dựng Priority Queue.

3.3 Binary Heap

Định nghĩa

Binary Heap là một cây nhị phân hoàn chỉnh được sử dụng để lưu trữ dữ liệu một cách hiệu quả nhằm lấy được phần tử lớn nhất hoặc nhỏ nhất dựa trên cấu trúc của nó.



Ví dụ về Binary Heap.

Binary Heap thường được biểu diễn dưới dạng mảng.

Phân loại Binary Heap

Max-Heap: Cho dãy số $h_0, h_1, \ldots, h_{n-1}$ là một max-heap nếu thỏa mãn:

$$\begin{cases} h_i \geq h_{2i+1} \\ h_i \geq h_{2i+2} \end{cases}, \forall i \in \left[0, \left[\frac{n}{2}\right] - 1\right]$$

Min-Heap: Cho dãy số $h_0, h_1, \ldots, h_{n-1}$ là một min-heap nếu thỏa mãn:

$$\begin{cases} h_i \leq h_{2i+1} \\ h_i \leq h_{2i+2} \end{cases}, \forall i \in \left[0, \left[\frac{n}{2}\right] - 1\right]$$

Một số thao tác trên Binary Heap

Ở đây, ta xét Binary Heap này là một Min Heap, các theo tác trên Max Heap cũng tương tự như vậy. Binary Heap được xét ở đây đang lưu trong mảng và chỉ số được bắt đầu từ 0.

3.4 Ứng dụng của Binary Heap

Heap Sort

Dưới đây là mã giả thuật toán Heap Sort sắp xếp một mảng a tăng dần:

```
function heapify(a[], l, r):
    i = l
    j = 2 * i + 1
    while j <= r:
        if j < r and a[j + 1] > a[j]:
            j++
        if a[j] <= a[i]:
            break
        swap(a[i], a[j]);
        i = j
        j = 2 * i +

function heapSort(a[], n):
    k = n / 2
    while (k >= 0):
```

```
16          heapify(a, k--, n - 1)
17      k = n - 1
18      while (k):
19          swap(a[0], a[k--])
20          heapify(a, 0, k)
```

Priority Queue

Ứng dụng của Binary Heap ở trong phần Priority Queue sẽ được trình bày ở mục tiếp theo.

4 Priority Queue

Ở trong mục này, ta xét đang làm việc với hàng đợi ưu tiên có phần tử ở đỉnh hàng đợi là lớn nhất hay nó chính là một Max-Heap. Các thao tác làm việc với Min-Heap tương tự.

4.1 Minh họa cách thức hoạt động của Priority Queue

Phần này sẽ minh họa các thao tác chính trong Priority Queue, bao gồm: **Heapify Up**, **Pop**, và **Heapify Down**.

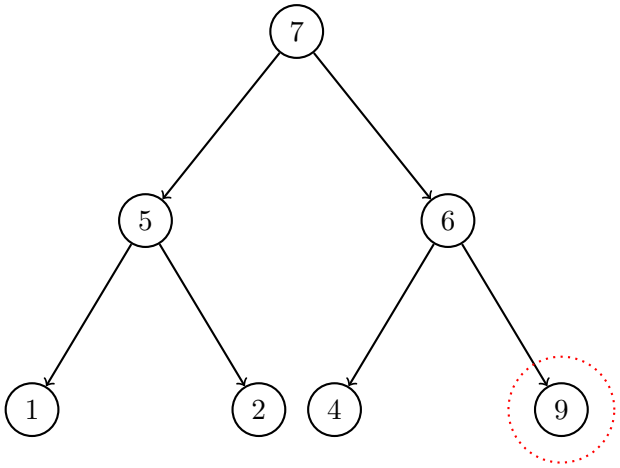
1. Thao tác Heapify Up

Khi thêm phần tử mới, ta sử dụng thao tác **Heapify Up** để đảm bảo tính chất Max-Heap.

Thêm đỉnh mới với giá trị mới vào vị trí phù hợp. Sau đó, lặp lại các bước so sánh giá trị giữa đỉnh đó và đỉnh cha, nếu chưa đảm bảo tính chất Max-Heap thì ta đổi giá trị giữa hai đỉnh và tiếp tục với đỉnh trên.

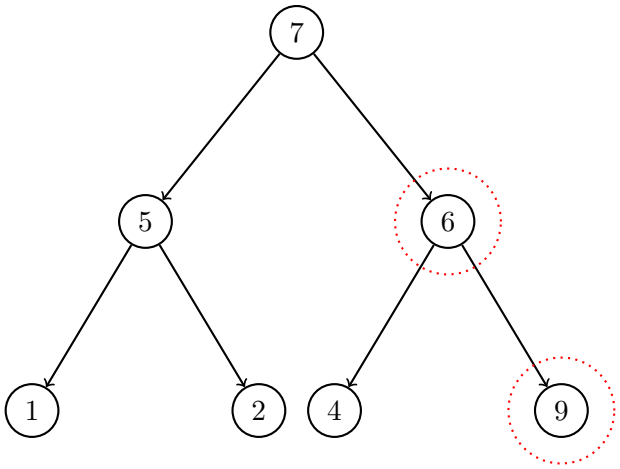
Quá trình này lặp lại cho đến khi đỉnh hiện tại đã đảm bảo tính chất Max-Heap hoặc nó nổi lên trở thành đỉnh gốc.

Cây ban đầu (sau khi thêm phần tử '9'):

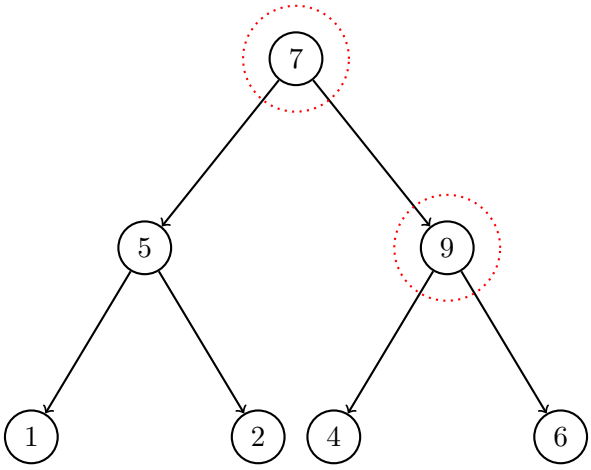


Max-Heap sau khi thêm phần tử mới '9'.

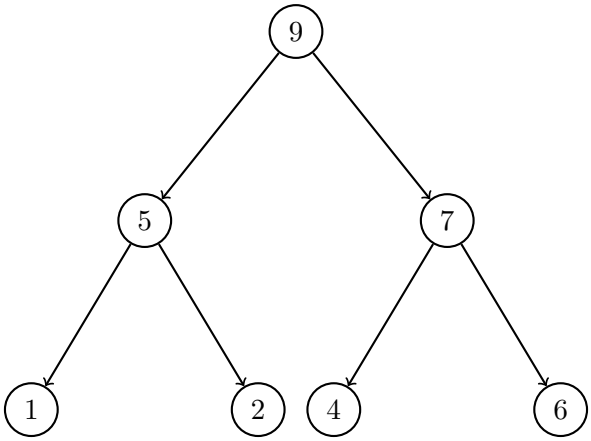
Cây sau khi thực hiện *Heapify Up* ('9' nổi lên):



So sánh '9' và cha của nó là '6', nếu lớn hơn thì đổi chỗ.



So sánh '9' và cha của nó hiện tại là '7', nếu lớn hơn thì đổi chỗ.

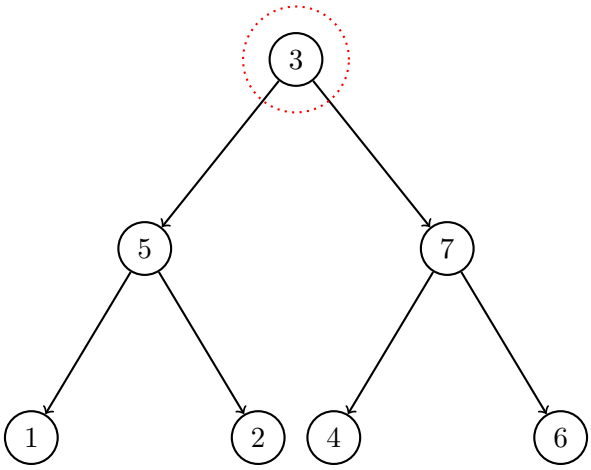


Lặp lại cho đến khi nó là đỉnh gốc hoặc không thỏa điều kiện lớn hơn.

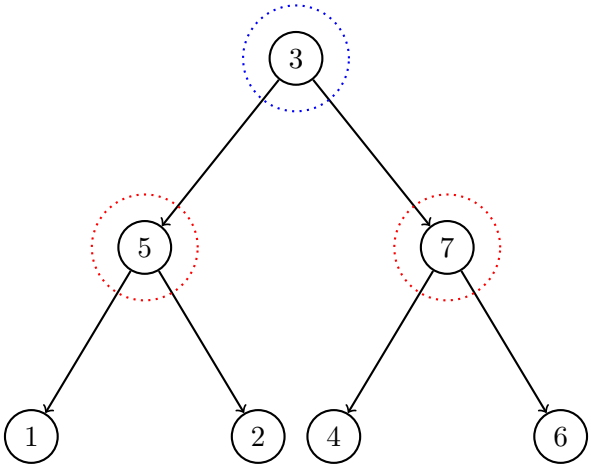
2. Thao tác Heapify Down

Tương tự như **Heapify Up**, trong trường hợp giá trị một phần tử thay đổi (ví dụ: '9' được thay bằng '3'), ta sử dụng thao tác **Heapify Down** để đưa phần tử này "chìm xuống" đúng vị trí.

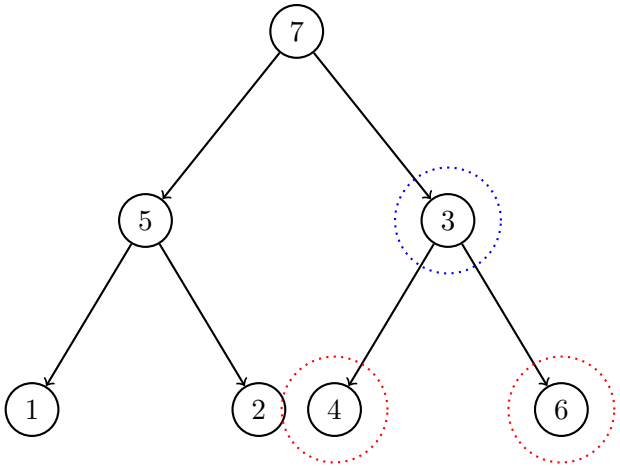
Cây ban đầu trước khi thay đổi giá trị ('9' -> '3'):



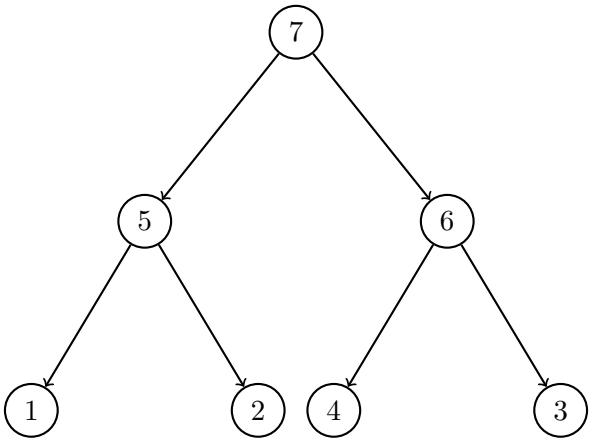
Cây sau khi **Heapify Down** ('3' chìm xuống):



So sánh đỉnh hiện tại với hai con của nó, nếu giá trị lớn nhất giữa hai đỉnh con lớn hơn đỉnh hiện tại thì đổi chỗ đỉnh hiện tại cho đỉnh lớn hơn ấy (Max-Heap).



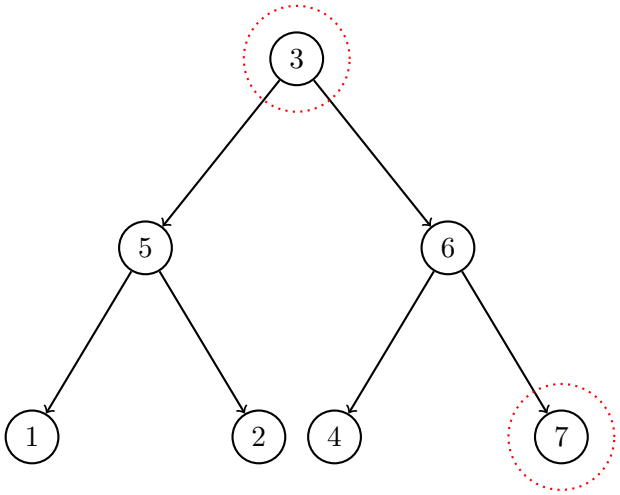
Tiếp tục lặp lại cho đến khi không còn thỏa điều kiện hoặc nó đã là đỉnh lá.



3. Thao tác Pop

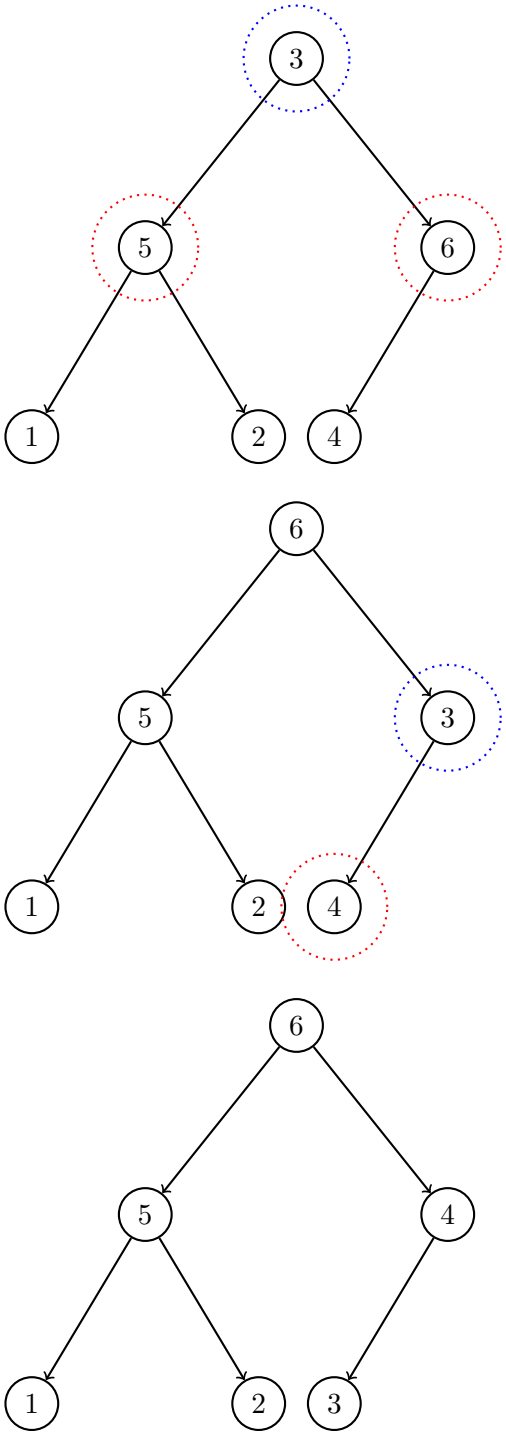
Khi xóa phần tử ưu tiên nhất (phần tử gốc), ta thay thế phần tử gốc bằng phần tử cuối cùng, sau đó sử dụng thao tác **Heapify Down** để duy trì tính chất Max-Heap.

Cây ban đầu trước khi Pop (‘7’ bị xóa nên ta hoán đổi với đỉnh lá ‘3’):



Max-Heap trước khi xóa phần tử lớn nhất 7

Cây sau khi Pop (xóa đỉnh '7' và **Heapify Down**):



Kết luận: Các thao tác **Heapify Up**, **Pop**, và **Heapify Down** đảm bảo Priority Queue hoạt động đúng, duy trì tính chất của Max-Heap trong mọi trường hợp.

4.2 Priority Queue xây dựng bằng danh sách liên kết

Cấu trúc này sử dụng các nút được liên kết bằng con trỏ. Mỗi nút chứa giá trị, con trỏ đến cha, con trái và con phải.

Kiểu dữ liệu ta sẽ làm việc

```
struct Node {
    value
    Node *parent
    Node *left
    Node *right
};

struct PriorityQueue {
    root = null
    last = null
    size = 0
};
```

heapifyUp

```
function heapifyUp(node):
    while node.parent != null and node.parent.value < node.value:
        swap(node.value, node.parent.value)
        node = node.parent
```

Đánh giá độ phức tạp của thuật toán:

Gọi n là kích thước của cây. Gọi h là chiều cao của cây.

Do cây này là một cây nhị phân hoàn chỉnh nên $h = \lfloor \log_2 n \rfloor$.

Xét trường hợp nút đang xét có giá trị lớn nhất trong max-heap.

Khi đó, sau khi chạy xong hàm **heapifyUp**, nút này phải nằm ở trên đỉnh của max-heap.

Do đó độ phức tạp trong trường hợp này là $O(h)$ hay $O(\log n)$.

Vậy độ phức tạp của hàm **heapifyUp** là $O(\log n)$

heapifyDown

```
function heapifyDown(node):
    while node != null:
        largest = node
        if node.left != null and
           node.left.value > largest.value:
            largest = node.left
        if node.right != null and
           node.right.value > largest.value:
            largest = node.right
        if largest == node:
            break
        swap(node.value, largest.value)
        node = largest
```

Đánh giá độ phức tạp:

Nhận xét: Cho một cây có n nút. Kích thước của cây con có tối đa là $\frac{2n}{3}$ nút.

Chứng minh:

Gọi H là cây nhị phân hoàn hảo cấp h.

Khi đó $|H| = 2^{h+1} - 1$ nút.

Khi đó, cây con trái của H là L có $2^h - 1$ nút.

Ta thêm vào mỗi nút lá ở cây con trái của H thêm hai nút con. Khi đó:

$$\begin{cases} L = 2^h + 2^h - 1 = 2^{h+1} - 1 \\ H = 2^{h+1} - 1 + 2^h = 2^{h+1} + 2^h - 1 \end{cases}$$

Ta thấy:

$$\frac{L}{H} = \frac{2^{h+1} - 1}{2^{h+1} + 2^h - 1} = \frac{2.2^h - 1}{3.2^h - 1} < \frac{2}{3}$$

Do đó, ta được điều phải chứng minh.

Quay lại việc chứng minh độ phức tạp:

Gọi $T(x)$ là thời gian chạy trong trường hợp xấu nhất của thuật toán khi áp dụng cho một cây con có kích thước là x.

Đối với một cây có gốc tại một nút i, thời gian chạy là $\theta(1)$ để sửa lại các mối quan hệ

giữa các phần tử $i \to key; i \to left \to key; i \to right \to key$, cộng với thời gian chạy để thực thi **heapifyDown** trên cây con có gốc là một trong các con của nút thứ i.
Kích thước của cây con của nút x tối đa là $\frac{2x}{3}$, vì vậy chúng ta có thể biểu diễn thời gian chạy của **heapifyDown** bằng phương pháp truy hồi là:

$$\begin{cases} T(x) = 0, 0 < x \leq 1 \\ T(x) \leq T\left(\frac{2x}{3}\right) + \theta(1), x > 1 \end{cases}$$

Không mất tính tổng quát, xét $n \in N$ và $\left(\frac{3}{2}\right)^{k-1} < n \leq \left(\frac{3}{2}\right)^{k}$

$$\begin{aligned} T(n) &\leq T\left(\left(\frac{3}{2}\right)^{k}\right) \leq T\left(\left(\frac{3}{2}\right)^{k-1}\right) + 2\theta(1) \\ &\leq \ldots \leq T(1) + (k+1)\theta(1) = (k+1)\theta(1) \end{aligned}$$

Mà do, $\left(\frac{3}{2}\right)^{k-1} < n \to k < \log_{\frac{3}{2}} n + 1$. Suy ra: $T(n) < \left(\log_{\frac{3}{2}} n + 2\right)\theta(1)$. hay $T(n) \in O(\log n)$.
Vậy độ phức tạp của hàm **heapifyUp** là $O(\log n)$

Lấy phần tử có độ ưu tiên lớn nhất ra (top)

```
function top():
    if root == null:
        throw Error("Heap is empty")
    return root.value
```

Thêm phần tử (push)

1. Tạo nút mới với giá trị x.
2. Nếu heap rỗng, đặt nút này làm gốc (root).
3. Nếu không, tìm vị trí chèn tiếp theo trong cây. Ở đây, dựa trên kích cỡ của heap sau khi đưa về dạng nhị phân. Khi đó, kích cỡ trên là thứ tự duyệt ***level-order***.
    - Nếu chưa có con trái, gán nút mới vào con trái.
    - Nếu chưa có con phải, gán nút mới vào con phải.
4. Thiết lập quan hệ cha-con giữa nút mới và nút cha.
5. Duy trì tính chất heap bằng cách **heapifyUp**.

```
function createNode(value):
    node = new Node
    node.value = value
    node.left = node.right = node.parent = null
    return node

function binary_presentation(number):
    result = ""
    while number > 0:
        bit = number % 2
        result = to_string(bit) + result
        number = number / 2
    return result

function findInsertionPoint():
    path = binary_representation(size + 1)
    path = path.pop_front();
    dummy = root

    for direction in path[0 : len(path) - 2]:
        if direction == '0':
            dummy = dummy.left
        else:
            dummy = dummy.right

    return dummy

function push(x):
    newNode = createNode(x)
    if root == null:
        root = last = newNode
    else:
        parent = findInsertionPoint()
        if parent.left == null:
```

```
            parent.left = newNode
        else:
            parent.right = newNode
        newNode.parent = parent

    last = newNode
    size = size + 1
    heapifyUp(newNode)
```

Đánh giá độ phức tạp của thuật toán:

Ta thấy hàm **binary_presentation** là hàm chuyển đổi từ số thập phân sang số nhị phân. Do đó có độ phức tạp là $O(\log n)$. Nhờ vậy, ta cũng có được hàm **findInsertionPoint** có độ phức tạp $O(\log n)$.

Hàm **push** này gọi lại hàm **findInsertionPoint** và hàm **heapifyUp**. Mà hai hàm này đều có độ phức tạp là $O(\log n)$.

Vì vậy, hàm **push** có độ phức tạp là: $O(\log n)$.

Xóa phần tử lớn nhất (pop)

1. Hoán đổi giá trị của gốc và nút cuối cùng.
2. Xóa nút cuối cùng khỏi cây.
3. Cập nhật lại nút cuối cùng mới.
4. Duy trì tính chất heap bằng cách **heapifyDown**.

```
function deleteLastNode():
    parent = last.parent
    if parent.right == last:
        parent.right = null
    else:
        parent.left  = null
    delete last

function findLastNode():
    path = binary_representation(size)
    path = path.pop_front();
```

```
12      dummy = root
13
14      for direction in path[0 : len(path) - 1]:
15          if direction == '0':
16              dummy = dummy.left
17          else:
18              dummy = dummy.right
19
20      return dummy
21
22  function pop():
23      if root == null:
24          throw Error("Heap is empty")
25      if size == 1:
26          delete root
27          root = last = null
28          size--
29      else:
30          swap(root.value, last.value)
31          deleteLastNode()
32          size--
33          if size > 1:
34              last = findLastNode(size)
35          else:
36              last = root
37          heapifyDown(root)
```

Đánh giá độ phức tạp của thuật toán:

Ta thấy hàm **binary_presentation** là hàm chuyển đổi từ số thập phân sang số nhị phân. Do đó có độ phức tạp là $O(\log n)$. Nhờ vậy, ta cũng có được hàm **findLastNode** có độ phức tạp $O(\log n)$.

Hàm **pop** này gọi lại hàm **findLastNode** và hàm **heapifyDown**. Mà hai hàm này đều có độ phức tạp là $O(\log n)$.

Vì vậy, hàm **pop** có độ phức tạp là: $O(\log n)$.

4.3 Priority Queue xây dựng bằng mảng

Việc xây dựng Priority Queue bằng danh sách liên kết dễ thấy sẽ tốn rất nhiều bộ nhớ. Đi kèm với nó còn là vấn đề truy xuất dữ liệu, cập nhật và xử lý trên con trỏ có phần phức tạp và dễ gặp phải nhiều sai xót.

Bởi vì Binary Heap có tính chất là một Cây nhị phân hoàn chỉnh, nên ta có thể biểu diễn nó đơn giản dưới dạng mảng tuyến tính:

Với nút chỉ số i, ta có:

- Cây con trái: $2 * i + 1$.
- Cây con phải: $2 * i + 2$.
- Cha: $(i - 1)/2$

Với cách quản lý như trên, các vấn đề về con trỏ đã được loại bỏ.

heapifyUp

```
function heapifyUp(index):
    parent = (index - 1) / 2
    while index != 0 and heap[parent] < heap[index]:
        swap(heap[parent], heap[index])

        index  = parent
        parent = (index - 1) / 2
```

Đánh giá độ phức tạp của thuật toán:

Gọi $T(n)$ là thời gian chạy trong trường hợp xấu nhất của thuật toán khi áp dụng cho giá trị đầu vào là n. Gọi $A(i)$ là giá trị của index tại vòng lặp thứ i ở trong vòng while. Theo mã giả ở trên, ta có công thức truy hồi:

$$\begin{cases} A(0) = n \\ A(i) = \left\lfloor \dfrac{A(i-1)-1}{2} \right\rfloor, \quad \forall i \geq 1 \end{cases}$$

Suy ra:

$$A(i) \leq \frac{A(i-1)-1}{2}$$
$$\Rightarrow A(i)+1 \leq \frac{A(i-1)+1}{2} \leq \cdots \leq \left(\frac{1}{2}\right)^i (A(0)+1) = \left(\frac{1}{2}\right)^i (n+1), \quad \forall i \geq 1$$

Vì đang xét trong trường hợp xấu nhất nên vòng lặp sẽ dừng lại khi có i_0 sao cho $A(i_0) = 0$. Gọi k là giá trị đầu tiên mà $A(k) = 0$. Tức lúc này vòng lặp while đã chạy được k lần.

Áp dụng bất đẳng thức trên, ta được:

$$A(k) + 1 \leq \left(\frac{1}{2}\right)^k (n+1) \Rightarrow 1 \leq \left(\frac{1}{2}\right)^k (n+1) \Rightarrow k \leq \log_2(n+1)$$

Do đó, số vòng lặp ở while trong trường hợp xấu nhất của hàm bé hơn hoặc bằng $\log_2(n+1)$.

Suy ra: $T(n) \leq C \log_2 n$.

Vậy độ phức tạp của hàm **heapifyDown** ở đây là $O(\log n)$.

heapifyDown

```
function heapifyDown(index):
    largest = index
    left    = index * 2 + 1
    right   = index * 2 + 2
    if left  < heap.size() and heap[left]  > heap[largest]:
        largest = left
    if right < heap.size() and heap[right] > heap[largest]:
        largest = right

    if (largest != index):
        swap(heap[largest], heap[index])
        heapifyDown(largest)
```

Đánh giá độ phức tạp của thuật toán:

Độ phức tạp thuật toán của hàm này được đánh giá tương tự như hàm **heapidyDown** của linked list như đã trình bày ở trên.

Độ phức tạp của hàm này là: $O(\log n)$

Lấy phần tử có độ ưu tiên lớn nhất ra (top)

```
function top():
```

```
    if heap.empty():
        throw Error("Heap is empty")
    return heap.front()
```

Thêm phần tử (push)

1. Đẩy giá trị x vào cuối mảng heap.
2. Duy trì tính chất heap bằng cách **heapifyUp**.

```
function push(x):
    heap.push_back(x)
    heapifyUp(heap.size() - 1)
```

Đánh giá độ phức tạp của thuật toán:

Gọi kích thước của heap.size là n.

Khi đó, thao tác **heapifyUp** có độ phức tạp là $O(\log n)$.

Vì vậy, hàm này có độ phức tạp $O(\log n)$.

Xóa phần tử lớn nhất (pop)

1. Hoán đổi giá trị của gốc $heap.front()$ và nút cuối cùng $heap.back()$.
2. Xóa phần tử cuối cùng khỏi mảng.
3. Duy trì tính chất heap bằng cách **heapifyDown**.

```
function pop():
    if heap.empty():
        throw Error("Heap is empty")

    heap.front() = heap.back()
    heap.pop_back()
    heapifyDown(0)
```

Đánh giá độ phức tạp thuật toán:

Gọi kích thước của heap.size là n.

Khi đó, thao tác **heapifyDown** có độ phức tạp là $O(\log n)$.

Vì vậy hàm này có độ phức tạp $O(\log n)$.

5 Một số biến thể của Binary Heap

Binary heap là một cấu trúc dữ liệu phổ biến, cơ bản để quản lý hàng đợi ưu tiên. Dựa trên nguyên lý của Binary Heap, hai biến thể nổi bật là Binomial Heap và Fibonacci Heap đã được phát trển nhằm tối ưu hóa các thao tạp như hợp nhất (merge), giảm khóa, (decrease-key) và chèn khóa (insert). Ở phần này chúng ta sẽ tìm hiểu về ưu điểm, nhược điểm của hai biến thể này và so sánh độ hiệu quả của nó so với Binary Heap.

5.1 Binomial Heap

5.1.1 Cấu trúc và đặc điểm

- Cấu trúc: Binomial Heap là một tập hợp các cây nhị phân binomial. Một Binomial Tree cấp k có:
  - 2^k nút.
  - Độ cao k.
  - Nút gốc có k cây con là các Binomial Tree cấp $k-1, k-2, ..., 0$.
- Thuộc tính heap: Nút cha trong mỗi cây luôn nhỏ hơn hoặc bằng (Min-Heap) hoặc lớn hơn hoặc bằng (Max-Heap) nút con.

5.1.2 Ưu điểm

- Hợp nhất nhanh ($O(\log n)$).
- Phù hợp với hệ thống yêu cầu quản lý bộ nhớ hoặc các thuật toán phân tán.

5.1.3 Nhược điểm

- Truy xuất giá trị nhỏ nhất chậm hơn Binary Heap ($O(\log n)$)

5.2 Fibonacci Heap

5.2.1 Cấu trúc và đặc điểm

- Cấu trúc: Fibonacci Heap là một tập hợp các cây (không cần cân bằng), liên kết với nhau bằng danh sách liên kết.

- Thuộc tính heap: Gốc của heap chứa giá trị nhỏ nhất (Min-Heap) hoặc lớn nhất (Max-Heap).
- Lazy structure: Nhiều thao tác được trì hoãn (deferred), giúp giảm độ phức tạp trung bình.

5.2.2 Ưu điểm

- Chèn khóa và giảm khóa rất nhanh ($O(\log n)$).
- Phù hợp cho các thuật toán như Dijkstra và Prim, nơi cần giảm khóa thường xuyên.

5.2.3 Nhược điểm

- Cấu trúc phức tạp, khó triển khai.
- Hiệu năng thực tế không luôn vượt trội do chi phí ẩn tử các thao tác bị trì hoãn.

5.3 So sánh Binary Heap, Binomial Heap và Fibonacci Heap

Thao tác	**Binary Heap**	**Binomial Heap**	**Fibonacci Heap**
Truy xuất nhỏ nhất	$O(1)$	$O(\log n)$	$O(1)$
Chèn khóa	$O(\log n)$	$O(\log n)$	$O(1)$
Hợp nhất (merge)	$O(n)$	$O(\log n)$	$O(1)$
Giảm khóa	$O(\log n)$	$O(\log n)$	$O(1)$ trung bình
Xóa nút	$O(\log n)$	$O(\log n)$	$O(\log n)$

Table 1: So sánh độ phức tạp của ba loại Heap

References

[1] J. W. J. Williams. *Wikipedia Entry for J. W. J. Williams.*

Truy cập tại: https://en.wikipedia.org/wiki/J._W._J._Williams.

[2] Dangerous Play Blog. *Heap Sort Algorithm and Explanation.*

Truy cập tại: https://dangerousplay.github.io/blog/heap_sort/.

[3] Heapsort. *Wikipedia Entry for Heapsort.*

Truy cập tại: https://en.wikipedia.org/wiki/Heapsort.

[4] Robert W. Floyd. *Algorithm 245: Treesort 3.* Journal of the ACM, Volume 10, Issue 4, October 1963.

Truy cập tại: https://dl.acm.org/doi/pdf/10.1145/355588.365103.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L.Rivest and Clifford Stein. *Introduction to algorithm - four edition.* April 2022.

[6] Binomial Heap. *GeeksforGeeks Entry for Binomial Heap.*

Truy cập tại: https://www.geeksforgeeks.org/binomial-heap-2/.

[7] Fibonacci Heap. *GeeksforGeeks Entry for Fibonacci Heap.*

Truy cập tại: https://www.geeksforgeeks.org/fibonacci-heap-insertion-and-union/.