VÕ TIẾN
Thảo luận kiến thức CNTT trường BK về KHMT(CScience), KTMT(CEngineering)
https://www.facebook.com/groups/khmt.ktmt.cse.bku

Cấu Trúc Dữ Liệu và Giải Thuật (DSA)

DSA2 - HK242

QUẢN LÝ KHO HÀNG SỬ DỤNG HASH, HEAP, TREE, GRAPH

Thảo luận kiến thức CNTT trường BK
về KHMT(CScience), KTMT(CEngineering)
https://www.facebook.com/groups/khmt.ktmt.cse.bku

TP. HỒ CHÍ MINH, THÁNG 4/2025

Mục lục

1 Định Nghĩa

Heap là một cấu trúc dữ liệu dạng cây nhị phân gần hoàn chỉnh (binary tree near-complete), trong đó:

- Tất cả các cấp (level) của cây, trừ cấp cuối cùng, đều được lấp đầy hoàn toàn.
- Cấp cuối cùng được lấp đầy từ trái sang phải.
- Tính chất heap được duy trì: giá trị của node cha so với node con thỏa mãn một điều kiện nhất định.

Min-Heap

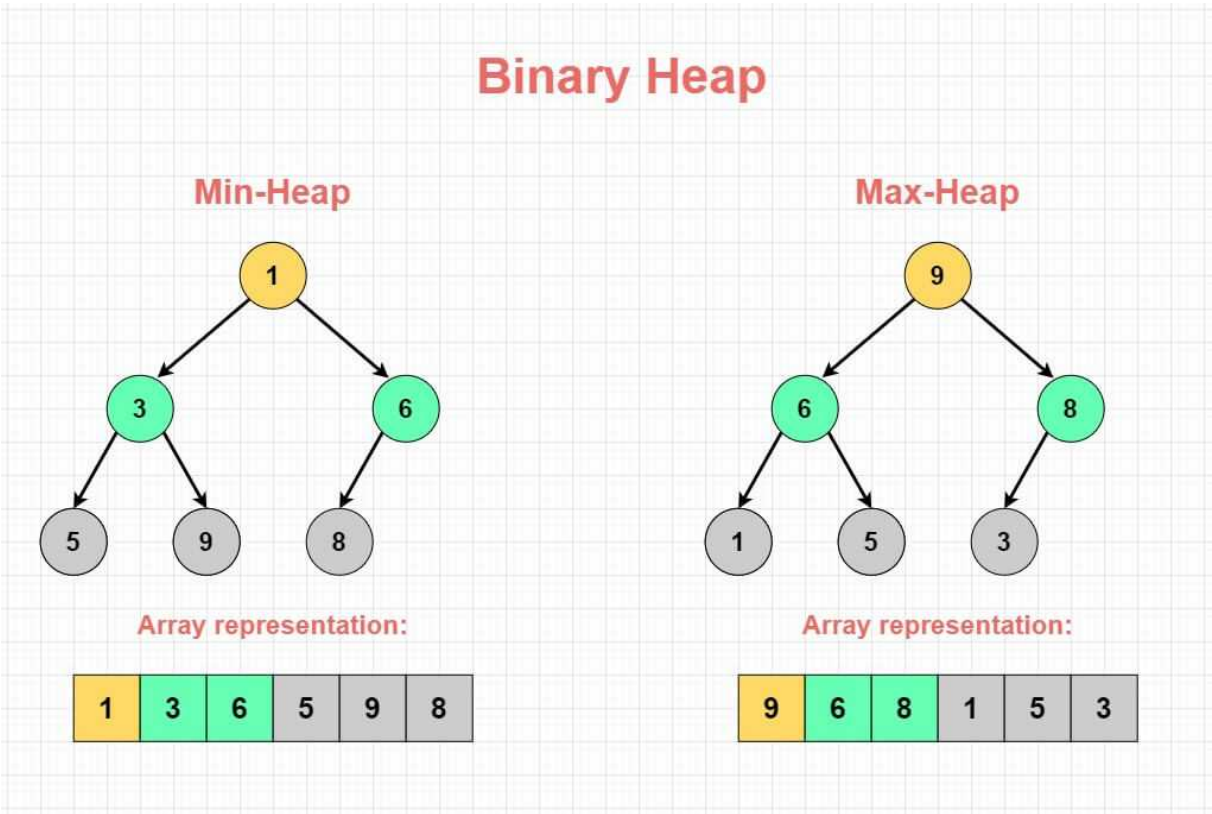
- Cây nhị phân gần hoàn chỉnh.
- Giá trị của node cha luôn **nhỏ hơn hoặc bằng** giá trị của node con:

  array[i] <= array[2*i + 1] (con trái) và array[i] <= array[2*i + 2] (con phải).
- Node gốc (array[0]) là giá trị **nhỏ nhất** trong heap.

Max-Heap

- Cây nhị phân gần hoàn chỉnh.
- Giá trị của node cha luôn **lớn hơn hoặc bằng** giá trị của node con:

  array[i] >= array[2*i + 1] (con trái) và array[i] >= array[2*i + 2] (con phải).
- Node gốc (array[0]) là giá trị **lớn nhất** trong heap.



2 Heap Algorithms

Heap tận dụng tính chất duyệt theo **chiều rộng (Breadth-First Search - BFS)**, tức là từ gốc xuống lá và từ trái sang phải, để ánh xạ các node của cây vào một mảng. Với cách này:

- **Array[0]** là node gốc (root), chứa giá trị **nhỏ nhất** (min-heap) hoặc **lớn nhất** (max-heap), tùy thuộc vào loại heap.
- Với mỗi node tại vị trí i trong mảng:
  - **Array[2*i + 1]**: Node con bên trái của Array[i].
  - **Array[2*i + 2]**: Node con bên phải của Array[i].
  - **Array[(i-1)/2]**: Node cha của Array[i] (với i $>$ 0, phép chia lấy nguyên).

Một số đặc điểm quan trọng

1. **Chiều cao của heap**:
   - Công thức: $h = \lfloor \log_2(n+1) \rfloor$, trong đó n là số phần tử trong mảng (tổng số node).
   - Điều này xuất phát từ việc heap là cây gần hoàn chỉnh, và chiều cao phụ thuộc vào số lượng node.
2. **Level tối đa**:
   - Level (cấp) của một node phụ thuộc vào vị trí của nó trong mảng và chiều cao h.
   - Ví dụ: node tại Array[i] có cấp là log(i + 1).

Ví dụ minh họa

Giả sử ta có một **min-heap** với các giá trị [3, 7, 5, 9, 10]:

- Mảng: [3, 7, 5, 9, 10].
- Cấu trúc:
  - Array[0] = 3 (gốc).
  - Con trái của 3: Array[2*0 + 1] = Array[1] = 7.
  - Con phải của 3: Array[2*0 + 2] = Array[2] = 5.
  - Cha của 7: Array[(1-1)/2] = Array[0] = 3.
- Kiểm tra: 3 <= 7 và 3 <= 5 (đúng với min-heap).

```cpp
template <typename E>
class Heap {
private:
    std::vector<E> data; //! dữ liệu
    bool compare(const E& a, const E& b){
        return a > b;     // min heap
        // return b > a;  // max heap
    }
    void heapifyUp_loop(int index);
    void heapifyUp_recursive(int index);
    void heapifyDown_loop(int index);
    void heapifyDown_recursive(int index);
public:
    void push(const E& value);
    void pop();
    void build(int* arr, int n);
    E top();
    bool isEmpty() const;
};
```

Giải thích các thành phần

1. **Thuộc tính private**:
   - std::vector<E> data: Lưu trữ các phần tử của heap dưới dạng mảng.
   - bool compare(const E& a, const E& b): Hàm so sánh để xác định loại heap.
     - return a > b: Min-heap (node cha nhỏ hơn node con).
     - return b > a: Max-heap (node cha lớn hơn node con).
2. **Các hàm private**:
   - heapifyUp_loop và heapifyUp_recursive: Điều chỉnh heap từ dưới lên (dùng khi thêm phần tử).
   - heapifyDown_loop và heapifyDown_recursive: Điều chỉnh heap từ trên xuống (dùng khi xóa phần tử).
3. **Các hàm public**:
   - push: Thêm một phần tử vào heap.
   - pop: Xóa phần tử gốc (nhỏ nhất hoặc lớn nhất).
   - build: Xây dựng heap từ một mảng đầu vào.
   - top: Trả về phần tử gốc.
   - isEmpty: Kiểm tra heap có rỗng không.
4. **heapifyUp_loop**
   - **Mục đích**: Điều chỉnh heap từ dưới lên (từ một node cụ thể lên gốc) khi thêm phần tử mới để đảm bảo tính chất heap.
   - **Chi tiết**:
     - Bắt đầu từ index (thường là phần tử cuối vừa thêm).
     - So sánh với node cha (parent = (index - 1) / 2).
     - Nếu cha không thỏa mãn (lớn hơn con trong min-heap), hoán đổi và tiếp tục lên trên.

- **Thuật toán**:

```
while (index > 0) {
    int parent = (index - 1) / 2;
    if (compare(data[parent], data[index])) { // Cha > Con
        std::swap(data[parent], data[index]);
        index = parent;
    } else {
        break;
    }
}
```

- **Ví dụ**: Thêm 2 vào min-heap [3, 7, 5]:
  - Ban đầu: data = [3, 7, 5, 2], index = 3.
  - Cha của 2: data[1] = 7, $7 > 2 \rightarrow$ hoán đổi: [3, 2, 5, 7].
  - Cha của 2: data[0] = 3, $3 > 2 \rightarrow$ hoán đổi: [2, 3, 5, 7].
  - Dừng vì index = 0.

5. **heapifyUp_recursive**

- **Mục đích**: Tương tự `heapifyUp_loop`, nhưng sử dụng đệ quy thay vì vòng lặp để điều chỉnh heap từ dưới lên.
- **Chi tiết**:
  - Kiểm tra nếu `index <= 0` thì dừng (đã tới gốc).
  - Tính node cha và so sánh, nếu cần hoán đổi thì gọi lại chính nó với vị trí cha.
- **Thuật toán**:

```
if (index <= 0) return;
int parent = (index - 1) / 2;
if (compare(data[parent], data[index])) {
    std::swap(data[parent], data[index]);
    heapifyUp_recursive(parent);
}
```

- **Ví dụ**: Thêm 2 vào min-heap [3, 7, 5]:
  - Ban đầu: data = [3, 7, 5, 2], index = 3.
  - Cha của 2: data[1] = 7, $7 > 2 \rightarrow$ hoán đổi: [3, 2, 5, 7], gọi lại với index = 1.
  - Cha của 2: data[0] = 3, $3 > 2 \rightarrow$ hoán đổi: [2, 3, 5, 7], gọi lại với index = 0.
  - Dừng vì index = 0.

6. **heapifyDown_loop**

- **Mục đích**: Điều chỉnh heap từ trên xuống (từ một node cụ thể xuống lá) để khôi phục tính chất heap, thường dùng khi xóa gốc hoặc xây dựng heap.
- **Chi tiết**:
  - Tìm node nhỏ nhất trong 3 node: chính nó, con trái, con phải.
  - Nếu node nhỏ nhất không phải nó, hoán đổi và tiếp tục xuống dưới.

- **Thuật toán**:

```
int size = data.size();
while (true) {
    int smallest = index;
    int left = 2 * index + 1;
    int right = 2 * index + 2;
    if (left < size && compare(data[smallest], data[left])) {
        smallest = left;
    }
    if (right < size && compare(data[smallest], data[right])) {
        smallest = right;
    }
    if (smallest == index) break;
    std::swap(data[index], data[smallest]);
    index = smallest;
}
```

- **Ví dụ**: Min-heap [7, 2, 5], gọi `heapifyDown_loop(0)`:
  - index = 0, data[0] = 7, con trái = 2, con phải = 5.
  - $7 > 2 \rightarrow$ smallest = 1, hoán đổi: [2, 7, 5].
  - Dừng vì không còn con nhỏ hơn.

7. **heapifyDown_recursive**
   - **Mục đích**: Tương tự `heapifyDown_loop`, nhưng dùng đệ quy để điều chỉnh heap từ trên xuống.
   - **Chi tiết**:
     - Tìm node nhỏ nhất, hoán đổi nếu cần, rồi gọi lại chính nó với vị trí mới.
   - **Thuật toán**:

```
int size = data.size();
int smallest = index;
int left = 2 * index + 1;
int right = 2 * index + 2;
if (left < size && compare(data[smallest], data[left])) {
    smallest = left;
}
if (right < size && compare(data[smallest], data[right])) {
    smallest = right;
}
if (smallest != index) {
    std::swap(data[index], data[smallest]);
    heapifyDown_recursive(smallest);
}
```

   - **Ví dụ**: Min-heap [7, 2, 5], gọi `heapifyDown_recursive(0)`:
     - index = 0, $7 > 2 \rightarrow$ hoán đổi: [2, 7, 5], gọi lại với index = 1.
     - index = 1, không hoán đổi nữa $\rightarrow$ dừng.

8. **push**

- **Mục đích**: Thêm một phần tử mới vào heap.
- **Chi tiết**:
  - Thêm phần tử vào cuối vector, sau đó gọi `heapifyUp` để điều chỉnh lên trên.
- **Thuật toán**:

```
data.push_back(value);
heapifyUp_loop(data.size() - 1); // Hoặc heapifyUp_recursive
```

- **Ví dụ**: Min-heap [3, 7, 5], `push(2)`:
  - Thêm: [3, 7, 5, 2].
  - `heapifyUp(3)` → [2, 3, 5, 7].

9. **pop**
   - **Mục đích**: Xóa phần tử gốc (nhỏ nhất trong min-heap).
   - **Chi tiết**:
     - Thay gốc bằng phần tử cuối, xóa phần tử cuối, rồi gọi `heapifyDown`.
   - **Thuật toán**:

```
if (isEmpty()) throw std::runtime_error("Heap is empty");
data[0] = data.back();
data.pop_back();
if (!isEmpty()) heapifyDown_loop(0); // Hoặc heapifyDown_recursive
```

   - **Ví dụ**: Min-heap [2, 3, 5, 7], `pop()`:
     - Thay 2 bằng 7: [7, 3, 5].
     - `heapifyDown(0)` → [3, 7, 5].
10. **build**
    - **Mục đích**: Xây dựng heap từ một mảng đầu vào.
    - **Chi tiết**:
      - Sao chép mảng vào vector, sau đó gọi `heapifyDown` từ node cha cuối cùng lên gốc.
    - **Thuật toán**:

```
data.assign(arr, arr + n);
for (int i = (n - 1) / 2; i >= 0; --i) {
    heapifyDown_loop(i); // Hoặc heapifyDown_recursive
}
```

    - **Ví dụ**: Mảng [4, 10, 3, 5, 1]:
      - Sao chép: [4, 10, 3, 5, 1].
      - Từ i = 1 (10): [4, 1, 3, 5, 10].
      - Từ i = 0 (4): [1, 4, 3, 5, 10].
11. **top**

- **Mục đích**: Trả về phần tử gốc mà không xóa.
- **Chi tiết**:
  - Chỉ lấy `data[0]`, kiểm tra rỗng trước.
- **Thuật toán**:

```
if (isEmpty()) throw std::runtime_error("Heap is empty");
return data[0];
```

- **Ví dụ**: Min-heap [1, 4, 3], `top()` trả về 1.

12. **isEmpty**
    - **Mục đích**: Kiểm tra heap có rỗng không.
    - **Chi tiết**:
      - Trả về `true` nếu vector rỗng.
    - **Thuật toán**:

```
return data.empty();
```

    - **Ví dụ**: [1, 4, 3] → `false`, [] → `true`.

3 Heap

3.1 Thuộc Tính Heap

1. **int capacity**: Sức chứa hiện tại của heap, khởi động mặc nhiên là 10.
2. **int count**: Số lượng phần tử hiện có trong heap.
3. **T* elements**: Mảng động lưu trữ các phần tử của heap.
4. **int (*comparator)(T& lhs, T& rhs)**: Con trỏ hàm so sánh hai phần tử kiểu T để xác định thứ tự của chúng trong heap.
   (a) Trường hợp **comparator** là NULL:
   - kiểu T phải hỗ trợ hai phép toán so sánh là > và <.
   - **Heap<T>** là một min-heap.

   (b) Trường hợp **comparator** khác NULL:
   - Muốn **Heap<T>** là max-heap thì hàm **comparator** trả về ba giá trị theo quy luật sau:
     – +1: nếu **lhs** < **rhs**.
     – -1: nếu **lhs** > **rhs**.
     – 0: trường hợp khác.
   - Muốn **Heap<T>** là min-heap thì hàm **comparator** trả về ba giá trị theo quy luật sau:
     – -1: nếu **lhs** < **rhs**.
     – +1: nếu **lhs** > **rhs**.
     – 0: trường hợp khác.

```
bool aLTb(T& a, T& b) { return compare(a, b) < 0; }
int compare(T& a, T& b) {
    if (comparator != 0)
        return comparator(a, b);
    else {
      if (a < b)
        return -1;
      else if (a > b)
        return 1;
      else
        return 0;
    }
}
```

Được dùng ở bước reheapUp và reheapDown hãy suy luận để dùng hàm aLTb hợp lí

5. **void (*deleteUserData)(Heap<T> pHeap)**: Con trỏ hàm dùng để giải phóng dữ liệu người dùng khi heap không còn được sử dụng. Trong trường hợp kiểu T là con trỏ và người dùng có nhu cầu để heap phải tự giải phóng bộ nhớ của các phần tử thì người dùng cần phải truyền một hàm cho **deleteUserData**, thông qua hàm khởi tạo. Người dùng cũng không cần định nghĩa hàm mới, chỉ cần truyền hàm **Heap<T>::free** vào hàm khởi tạo của **Heap<T>**. **(như BTL1 thôi khá là dễ)**

3.2 Các Hàm cần triển khai

3.2.1 Các hàm phụ được gọi từ các hàm trong chính được public ra ngoài

1. **void ensureCapacity(int minCapacity)**: Đảm bảo mảng động `elements` có đủ sức chứa tối thiểu yêu cầu.
2. **void swap(int a, int b)**: Hoán đổi vị trí của hai phần tử trong mảng `elements`.
3. **void reheapUp(int position)**: Điều chỉnh heap từ dưới lên trên để duy trì tính chất heap sau khi thêm phần tử mới.
4. **void reheapDown(int position)**: Điều chỉnh heap từ trên xuống dưới để duy trì tính chất heap sau khi xóa phần tử.
5. **int getItem(T item)**: Tìm và trả về chỉ số của phần tử `item` trong heap.
6. **void removeInternalData()**: Giải phóng dữ liệu trong mảng `elements` và các tài nguyên nếu cần.
7. **void copyFrom(const Heap<T>& heap)**: Sao chép dữ liệu từ một heap khác vào heap hiện tại.

3.2.2 Cách hàm chính

1. **Hàm khởi tạo**

```
Template <class T>
Heap<T>::Heap(int (*comparator)(T&, T&), void (*deleteUserData)(Heap<T>*)) {
    this->capacity = 10;  // Giá trị khởi tạo ban đầu
    this->count = 0;
    this->elements = new T[this->capacity];
    this->comparator = comparator;
    t
```

2. **hàm hủy**

```
template <class T>
Heap<T>::~Heap() {
    this->removeInternalData();
}
```

3. **Hàm push(T item)**
   (a) Gọi hàm `ensureCapacity` để đảm bảo mảng động `elements` có thể chứa `(count + 1)` phần tử.
   (b) Thêm phần tử vào vị trí cuối của mảng `elements`.
   (c) Thực hiện quá trình `reheapUp` để đưa phần tử về đúng vị trí, đảm bảo tính chất heap.
   (d) Tăng giá trị biến `count`.

```
template <class T>
void Heap<T>::push(T item) {
    ensureCapacity(count + 1);  // Đảm bảo đủ không gian
    elements[count] = item;     // Thêm phần tử mới vào cuối mảng
    reheapUp(count);            // Điều chỉnh vị trí để duy trì tính chất heap
```

```
    count++;                    // Tăng số lượng phần tử
}
```

4. **Hàm pop()**

   (a) Đưa phần tử ở cuối cùng trên mảng `elements` lên vị trí số 0, phải backup lại phần tử tại 0 để trả về.

   (b) Thực hiện quá trình `reheapDown` để duy trì tính chất heap.

   (c) Cập nhật biến `count`.

```
    template <class T>
T Heap<T>::pop() {
    if (count == 0) throw std::underflow_error("Calling to peek with the empty
     ↪  heap.");
    T root = elements[0];         // Lấy phần tử gốc
    elements[0] = elements[count - 1];  // Đặt phần tử cuối lên đầu
    count--;
    reheapDown(0);                // Điều chỉnh lại heap
    return root;
}
```

5. **Hàm peek()**

   (a) Trả về phần tử gốc mà không xóa nó khỏi heap.

```
    template <class T>
const T Heap<T>::peek() {
    if (count == 0) throw std::underflow_error("Calling to peek with the empty
     ↪  heap.");
    return elements[0];
}
```

6. **void remove(T item, void (*removeItemData)(T))**

   (a) Gọi phương thức `getItem` để tìm vị trí của phần tử `item` trong heap. Nếu không tìm thấy phần tử, thoát khỏi phương thức.

   (b) Nếu phần tử được tìm thấy tại vị trí `foundIdx` trên `elements`, thay thế phần tử tại vị trí `foundIdx` bằng phần tử cuối cùng trong heap `(elements[count - 1])`.

   (c) Giảm số lượng phần tử (`count`) đi 1.

   (d) Gọi phương thức `reheapDown` từ vị trí `foundIdx` để khôi phục tính chất của heap.

   (e) Nếu con trỏ hàm `removeItemData` được cung cấp, gọi hàm này để giải phóng bộ nhớ hoặc xử lý dữ liệu của phần tử đã bị xóa.

```
    template <class T>
void Heap<T>::remove(T item, void (*removeItemData)(T)) {
    int pos = getItem(item);
    if (pos == -1) return;  // Phần tử không tồn tại

    if (removeItemData != nullptr) {
        removeItemData(elements[pos]);
```

```
    }
    elements[pos] = elements[count - 1];  // Đặt phần tử cuối lên vị trí bị xóa
    count--;
    reheapDown(pos);                      // Điều chỉnh lại heap
}
```

7. **Hàm contains(T item)**

   (a) Kiểm tra xem heap có chứa phần tử `item` hay không.

```
template <class T>
bool Heap<T>::contains(T item) {
    return getItem(item) != -1;
}
```

8. **Hàm size()**

   (a) Trả về số lượng phần tử hiện có trong heap.

```
template <class T>
int Heap<T>::size() {
    return count;
}
```

9. **Hàm heapify(T array[], int size)**

   (a) Duyệt qua từng phần tử trong mảng `array`.

   (b) Gọi phương thức `push` để thêm từng phần tử vào heap và duy trì tính chất của heap.

```
    template <class T>
void Heap<T>::heapify(T array[], int size) {
    // TODO YOUR CODE IS HERE
}
```

10. **Hàm empty()**

    (a) Kiểm tra xem heap có rỗng hay không.

```
template <class T>
bool Heap<T>::empty() {
    return count == 0;
}
```

11. **Hàm clear()**

    (a) Xóa tất cả các phần tử trong heap và đặt heap về trạng thái rỗng ban đầu.

```
template <class T>
void Heap<T>::clear() {
    this->removeInternalData();  // Xóa dữ liệu
```

```
    this->count = 0;
    this->capacity = 10;
    this->elements = new T[this->capacity];
}
```

3.2.3 reheapUp và reheapDown

1. **Hàm reheapUp(int position)** Hàm này được sử dụng để điều chỉnh vị trí của một phần tử trong heap từ vị trí hiện tại lên đúng vị trí của nó để duy trì tính chất heap.

```
template <class T>
void Heap<T>::reheapUp(int position) {
  // TODO YOUR CODE IS HERE
}
```

   - Tính chỉ số của phần tử cha bằng công thức $(position - 1)/2$.
   - Vòng lặp tiếp tục cho đến khi phần tử không còn là phần tử gốc (`position > 0`) và phần tử tại vị trí hiện tại lớn hơn phần tử cha (điều kiện `aLTb`).
   - Nếu điều kiện đúng, hoán đổi vị trí giữa phần tử hiện tại và phần tử cha.
   - Cập nhật `position` thành chỉ số của phần tử cha.
   - Cập nhật chỉ số của phần tử cha cho lần lặp tiếp theo.
   - Khi vòng lặp kết thúc, phần tử hiện tại sẽ nằm ở vị trí đúng trong heap.

2. **Hàm reheapDown(int position)** Hàm này điều chỉnh vị trí của một phần tử trong heap từ vị trí hiện tại xuống đúng vị trí của nó để duy trì tính chất heap.

```
template <class T>
void Heap<T>::reheapDown(int position) {
  // TODO YOUR CODE IS HERE
}
```

   - Tính chỉ số của con trái và con phải dựa trên chỉ số của phần tử hiện tại.
   - Giả định rằng phần tử lớn nhất hiện tại là phần tử ở vị trí `position`.
   - Nếu con trái tồn tại và lớn hơn phần tử hiện tại, cập nhật `largest` thành chỉ số con trái.
   - Nếu con phải tồn tại và lớn hơn phần tử hiện tại hoặc con trái, cập nhật `largest` thành chỉ số con phải.
   - Nếu `largest` khác với `position`, hoán đổi vị trí giữa phần tử hiện tại và phần tử lớn nhất.
   - Gọi hàm `reheapDown` đệ quy để tiếp tục điều chỉnh từ vị trí `largest` cho đến khi heap được duy trì đúng.
   - Khi không còn cần điều chỉnh nữa, hàm sẽ kết thúc.

3.2.4 Ví dụ về comparator

Trường hợp Comparator là NULL (Min-Heap)

Giả sử chúng ta có một min-heap với các phần tử sau, biểu diễn dưới dạng mảng:

$$\text{Heap: } [20, 30, 25, 40, 35]$$

Thêm phần tử '15':

1. **Bước 1: Thêm phần tử**
   - Thêm '15' vào cuối heap:

$$[20, 30, 25, 40, 35, 15]$$

2. **Bước 2: Thực hiện reheapUp**
   - So sánh '15' với cha nó ('25' tại chỉ số 2):
     - $15 < 25$, hoán đổi:

$$[20, 30, 15, 40, 35, 25]$$

   - So sánh '15' với cha mới của nó ('20' tại chỉ số 0):
     - $15 < 20$, hoán đổi tiếp:

$$[15, 30, 20, 40, 35, 25]$$

Trường hợp Comparator khác NULL (Max-Heap)

Giả sử chúng ta có một max-heap với các phần tử sau:

$$\text{Heap: } [30, 20, 25, 10, 15]$$

Hàm so sánh:

```
int compare(int& lhs, int& rhs) {
    if (lhs < rhs) return +1; // Điều kiện cho max-heap
    if (lhs > rhs) return -1;
    return 0;
}
```

Thêm phần tử '35':

1. **Bước 1: Thêm phần tử**
   - Thêm '35' vào cuối heap:

$$[30, 20, 25, 10, 15, 35]$$

2. **Bước 2: Thực hiện reheapUp**
   - So sánh '35' với cha nó ('25' tại chỉ số 2):
     - $35 > 25$, hoán đổi:

$$[30, 20, 35, 10, 15, 25]$$

   - So sánh '35' với cha mới của nó ('30' tại chỉ số 0):
     - $35 > 30$, hoán đổi tiếp:

$$[35, 20, 30, 10, 15, 25]$$

Kết quả: Heap được điều chỉnh thành công.

Trường hợp Comparator khác NULL (Min-Heap)

Giả sử chúng ta có một min-heap với các phần tử sau:

$$\text{Heap: } [20, 30, 25, 40, 35]$$

Hàm so sánh:

```
int compare(int& lhs, int& rhs) {
    if (lhs < rhs) return -1; // Điều kiện cho min-heap
    if (lhs > rhs) return +1;
    return 0;
}
```

Thêm phần tử '15':

1. **Bước 1: Thêm phần tử**
   - Thêm '15' vào cuối heap:

$$[20, 30, 25, 40, 35, 15]$$

2. **Bước 2: Thực hiện reheapUp**
   - So sánh '15' với cha nó ('25' tại chỉ số 2):
     - $15 < 25$, hoán đổi:

$$[20, 30, 15, 40, 35, 25]$$

   - So sánh '15' với cha mới của nó ('20' tại chỉ số 0):
     - $15 < 20$, hoán đổi tiếp:

$$[15, 30, 20, 40, 35, 25]$$