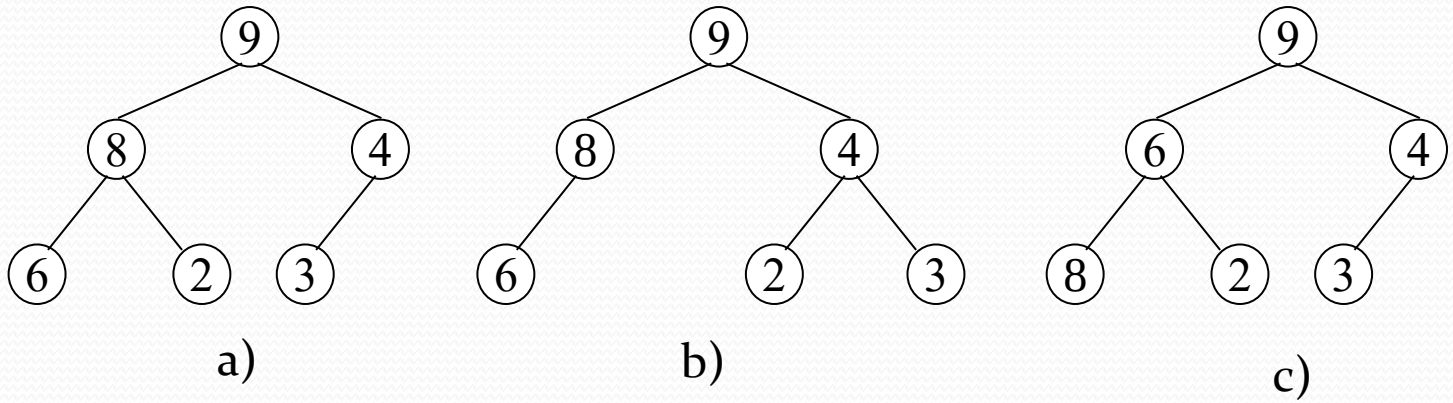
Cây vun đống (Heap)

Nội dung

- Khái niệm
- Tổ chức dữ liệu
- Thao tác trên Heap
- Ứng dụng
  - Heap Sort
  - Hàng đợi ưu tiên

Khái niệm

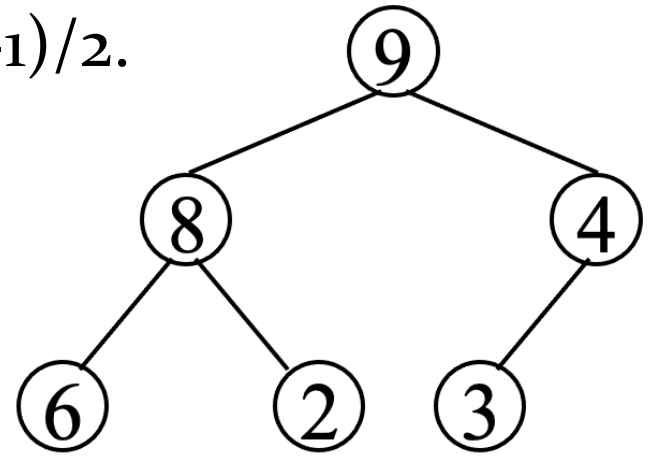
- Heap là một cây nhị phân đầy đủ mà mỗi nút của cây khóa của nút lớn hơn (hoặc nhỏ hơn) khóa của các nút con (nếu có).
- Ví dụ:
  - a) là heap, b) không là heap vì không phải cây nhị phân đầy đủ, c) không là heap vì vi phạm tính chất khóa



Tổ chức dữ liệu

- Dùng mảng: đánh số các nút của cây từ trên xuống dưới, từ trái qua phải bằng các số 0, 1,.., n-1.
- Mỗi nút lưu trong một ô nhớ của mảng theo thứ tự đã đánh số.
- Nút thứ i có 2 nút con là 2i+1 và 2i+2 và nút cha là (i-1)/2.

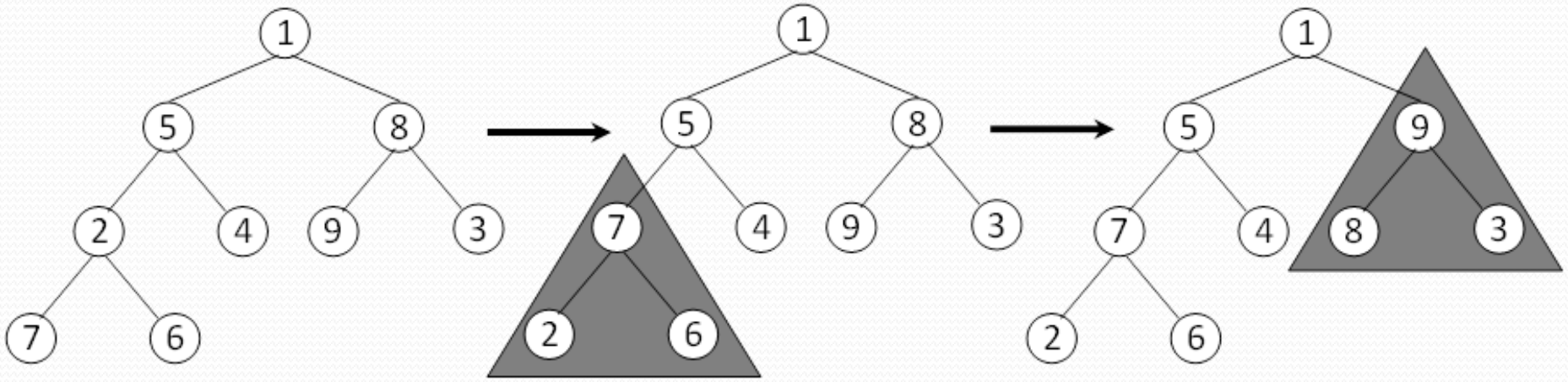
```
#define MAX_HEAP 1000
struct Heap
{
    ElementType arr[MAX_HEAP];
    int size;
};
```

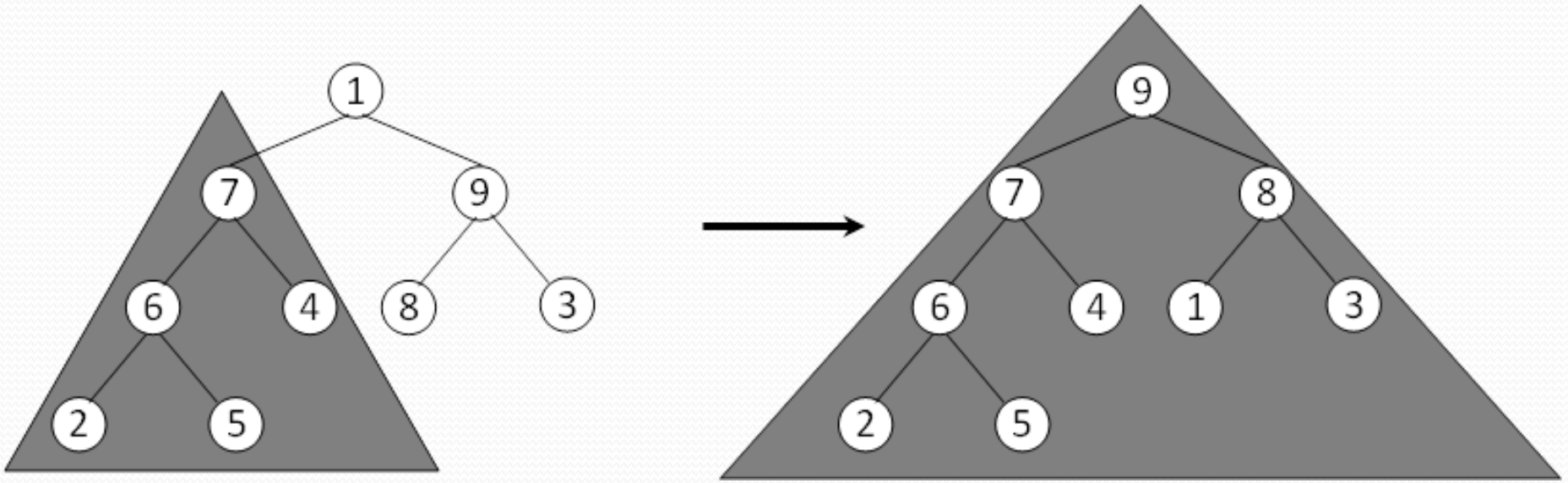


9	8	4	6	2	3				
0	1	2	3	4	5	6	7	8	9

Một số thuật toán cơ bản

- Thuật toán 1: chuyển cây nhị phân đầy đủ thành heap.
- Ví dụ:




1
7
9
6
4
8
3
2
5
9
7
8
6
4
1
3
2
5

Thuật toán SiftDown

Chuyển một cây nhị phân đầy đủ từ vị trí *r*..*n* trong mảng với các cây con trái và phải là các heap thành một heap.

Input: cây nhị phân đầy đủ thỏa điều kiện như trên

Output: cây nhị phân đầy đủ là heap

Action:

p = r

Lặp khi p còn nút con:

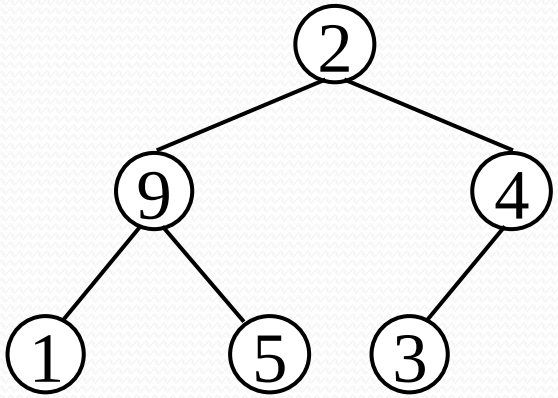
Tìm nút con c của p có khoá lớn nhất

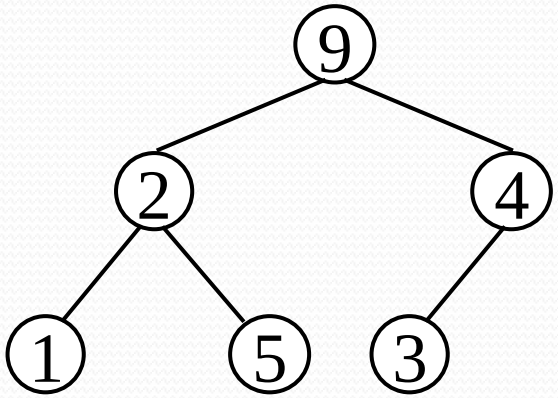
Nếu khóa của nút p nhỏ hơn khóa của c thì:

Đổi giá trị nút p với nút c

Chuyển p sang nút c

Ngược lại thì dừng





```
void siftDown(Heap &heap, int r, int n)
{
  int p, c; ElementType tmp;
  p = r; c = 2*p+1;
  while (c <= n)
  {
      if (c + 1 <= n && heap.arr[c].key < heap.arr[c+1].key)
            c = c + 1;
      if (heap.arr[p].key < heap.arr[c].key)
      {
        tmp = heap.arr[p];
        heap.arr[p] = heap.arr[c];
        heap.arr[c] = tmp;
        p = c;
        c = c*2 + 1;
      }
      else return;
   }
}
```

Độ phức tạp thuật toán

- Tại mỗi bước của thuật toán số phần tử cần thao tác giảm một nửa so với bước trước.
- Do đó độ phức tạp của thuật toán trong trường hợp xấu nhất là O(log$n$), với n là số phần tử của Heap.

Thuật toán tạo Heap

Input: cây nhị phân đầy đủ h

Output: h là một heap

Action:

Lặp i từ h.size/2-1 đến 0:

SiftDown(h, i, h.size-1)

```
void creatHeap(Heap &h)
{
  int i;
  for(i = h.size/2-1; i >= 0; i--)
     siftDown(h, i, h.size - 1);
}
```

Độ phức tạp thuật toán

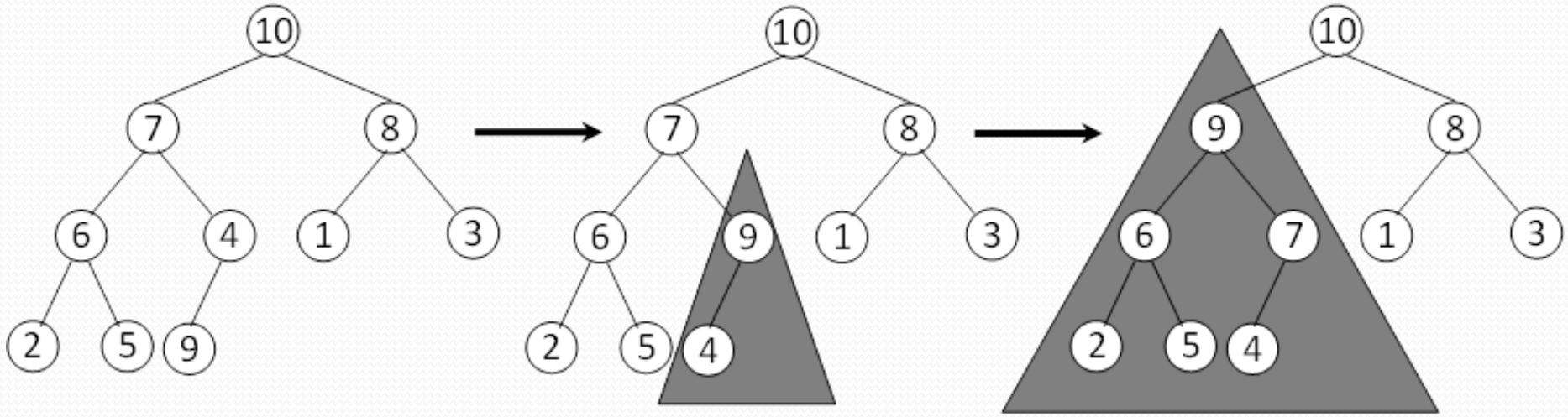
- Do siftDown có độ phức tạp tính toán là O(logn) nên thuật toán tạo heap có độ phức tạp tính toán là O(nlogn), với n là số phần tử của heap.

Hàng đợi ưu tiên

- Hàng đợi ưu tiên là một hàng đợi mà mỗi phần tử có một độ ưu tiên nhất định.
- Khi lấy ra thì phần tử nào có độ ưu tiên cao nhất được lấy ra trước. Những phần tử có cùng độ ưu tiên thì được lấy ra theo nguyên tắc vào trước ra trước.
- Dùng Heap để tổ chức dữ liệu cho hàng đợi ưu tiên

```
#define MAX_HEAP 1000
struct PriorityQueue
{
    ElementType arr[MAX_HEAP];
    int size;
};
```

- Thêm vào hàng đợi ưu tiên:
  - Đưa phần tử cần thêm vào cuối mảng của heap
  - Điều chỉnh mảng sau khi thêm thành heap
- Ví dụ: thêm số 9 vào heap



Thuật toán SiftUp

- Cho hàng đợi ưu tiên được lưu trên mảng từ vị trí 0 đến *n*-1, và là một Heap.
- Khi thêm phần tử x vào vị trí n thì mảng các phần tử có thể không còn là một Heap.
- Thuật toán SiftUp đổi các phần tử của mảng để trở thành Heap

Thuật toán SiftUp

Input: h có n+1 phần tử mà n phần tử đầu là heap

Output: h là heap

Action:

Xuất phát từ nút cuối cùng của cây

Lặp khi chưa đến nút gốc:

Nếu cây con chứa nút đang xét chưa tạo thành Heap thì:

Đổi nút đang xét cho nút gốc

Chuyển nút đang xét về nút gốc của cây con vừa điều chỉnh

Ngược lại thì dừng

- Cài đặt:

```
void siftUp(PriorityQueue &q)
{
  int i, p; ElementType tmp;
  i = q.size - 1;
  while(i > 0){
    p = (i-1)/2;
    if (q.arr[p].priority < q.arr[i].priority){
      tmp = q.arr[p];
      q.arr[p] = q.arr[i];
      q.arr[i] = tmp;
      i = p;
    }
    else
     return;
   }
}
```

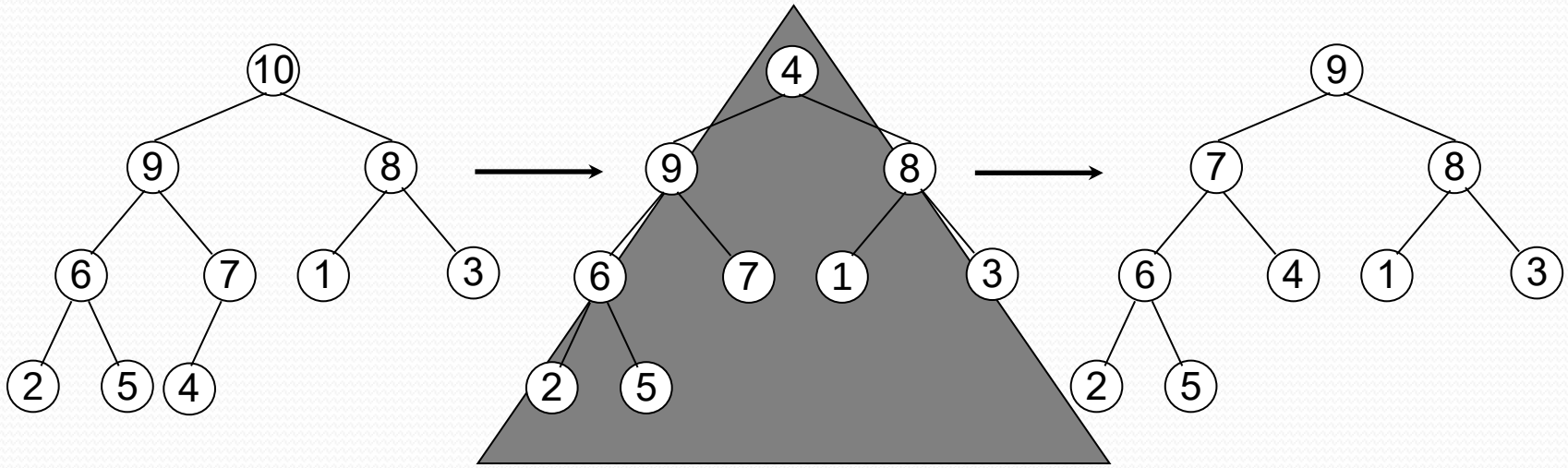
- Hàm thêm một phần tử vào hàng đợi ưu tiên

```
void add(PriorityQueue &q, ElementType x)
{
   q.arr[q.size] = x;
   q.size++;
   siftUp(q);
}
```

Độ phức tạp thuật toán

- Độ phức tạp tính toán của hàm thêm chính là độ phức tạp của thao tác SiftUp.
- Tương tự SiftDown, thao tác SiftUp có độ phức tạp là O(log*n*), với *n* là số phần tử của hàng đợi ưu tiên.

Thao tác lấy ra



- Thao tác lấy ra:
  - Cho hàng đợi ưu tiên có *n* phần tử được lưu trên mảng theo kiểu Heap.
  - Phần tử có độ ưu tiên cao nhất cần lấy ra chính là phần tử đầu tiên của mảng.
  - Để xoá phần tử ra khỏi hàng đợi ưu tiên mà vẫn thoả mãn các tính chất của Heap ta thực hiện 2 thao tác:
    - Chuyển phần tử cuối cùng của mảng (vị trí *n-1*) về đầu.
    - Dùng thao tác siftDown để chuyển *n*-1 phần tử của mảng thành một Heap.

- Cài đặt

```
ElementType remove(PriorityQueue &q)
{
   int n; ElementType x;
   n = q.size;
   x = q.arr[0];
   q.arr[0] = q.arr[n-1];
   q.size--;
   siftDown(q);
   return x;
}
```

```
void siftDown(PriorityQueue &q){
  int p = 0, i = 2*p + 1;
  while (i < q.size){
    if (i + 1 < q.size && q.arr[i].priority <
q.arr[i+1].priority)
      i = i + 1;
    if (q.arr[p].priority < q.arr[i].priority){
      ElementType tmp = q.arr[p];
      q.arr[p] = q.arr[i];
      q.arr[i] = tmp;
      p = i;
      i = 2*p + 1;
    }
    else
      return;
  }
}
```

Hàng đợi ưu tiên trong C++

- Lớp: `std:: priority_queue`
- Khai báo biến:
  - `std::priority_queue<DataType> variable;`
  - `std::priority_queue<DataType,vector<DataType>,compare>  variable;`
- Ví dụ:
  - `std::priority_queue<int> pq;`
  - `std::priority_queue<int, vector<int>, greater<int> > pq;`

- Ví dụ: hang đợi ưu tiên các sinh viên theo thứ tự giảm của điểm trung bình:

```
struct Student{
  string name;
  int age;
  float gpa;
};

struct CompareGPA {
   bool operator()(const Student& a, const Student& b) {
     return a.gpa > b.gpa;
   }
};

std::priority_queue<Student, vector<Student>, CompareGPA> pd;
```

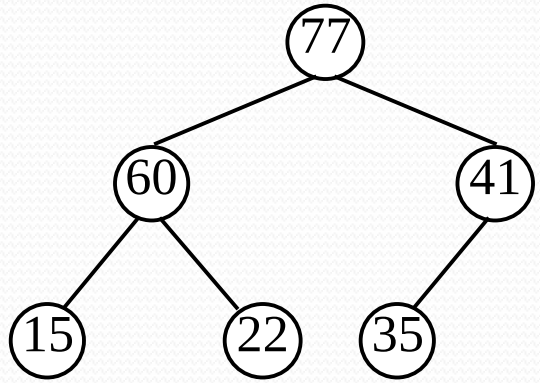
- Các phương thức:
- `push(element)`: thêm một phần tử vào hàng đợi ưu tiên
- `pop()`: xóa một phần tử có độ ưu tiên cao nhất
- `top()`: trả về phần tử có độ ưu tiên cao nhất
- `empty()`: kiểm tra hàng đợi ưu tiên có rỗng không
- `size()`: trả về số phần tử của hàng đợi ưu tiên.

```
priority_queue<int, vector<int>, greater<int>> pq;
```

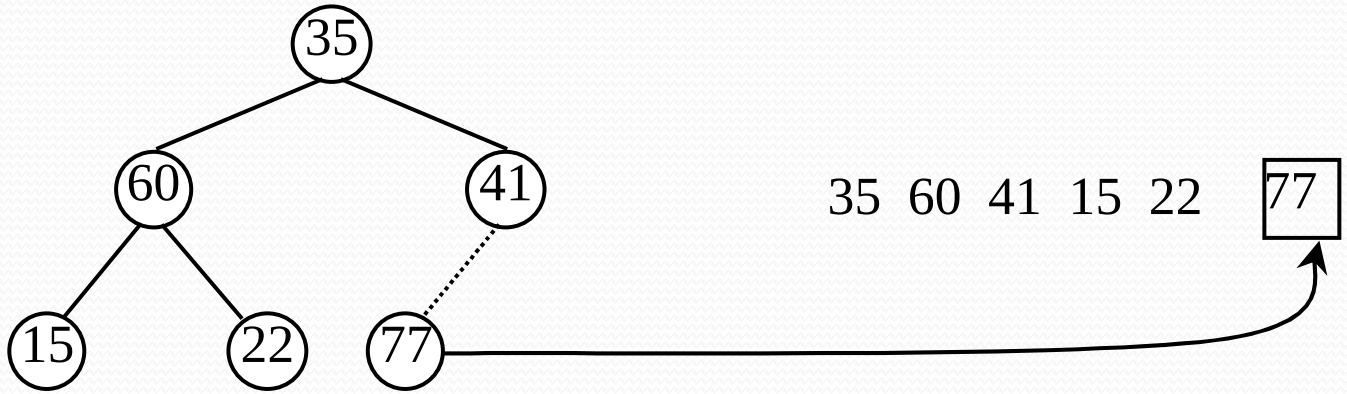
```
#include <queue>
#include <iostream>

using namespace std;

int main() {
  priority_queue<int> pq;

  pq.push(5);
  pq.push(2);
  pq.push(10);

  while (!pq.empty()) {
    cout << pq.top() << " ";
    pq.pop();
  }
  // output: 10 5 2
  return 0;
}
```

Thuật toán HeapSort

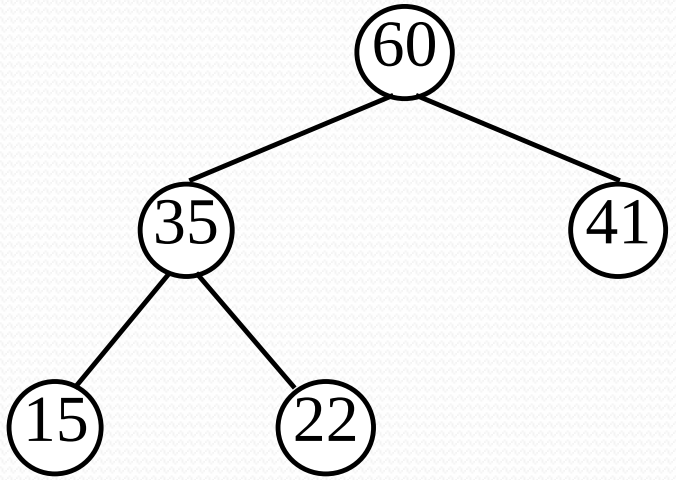
- Sắp xếp dãy số sau theo thứ tự tăng:
  35, 15, 77, 60, 22, 41.
- Tạo Heap từ mảng



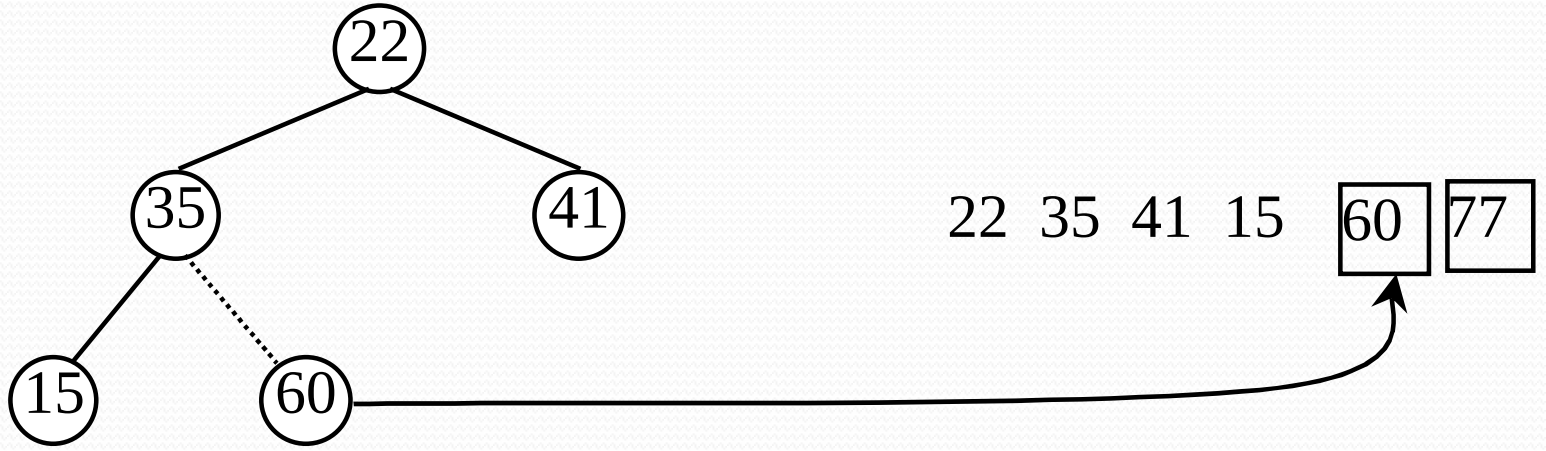
- Đổi chỗ



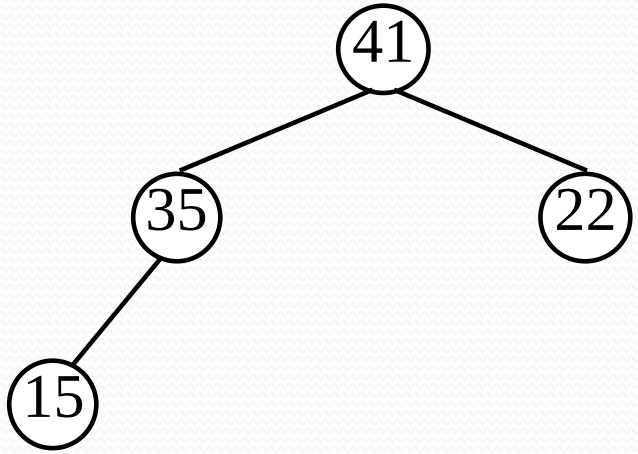
- Tạo điều chỉnh lại Heap



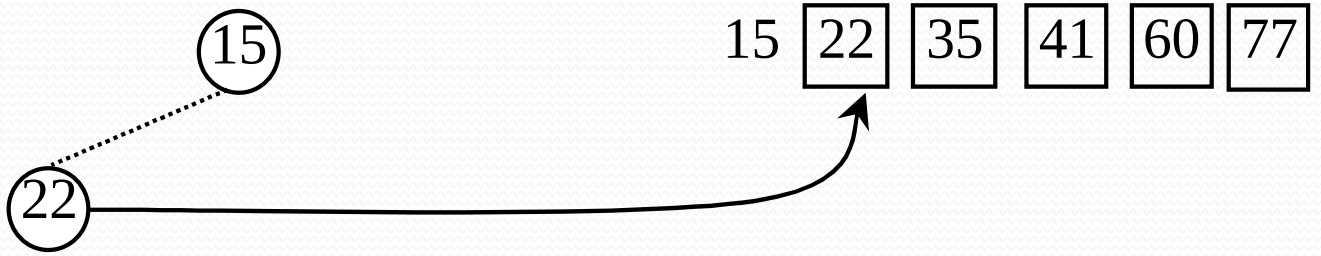
- Đổi chỗ



- Điều chỉnh



- Tiếp tục cho đến khi còn lại 1 số



Thuật toán HeapSort

- Sắp xếp mảng a[0..n-1] theo thứ tự tăng
- Dùng thao tác CreateHeap tạo Heap cho dãy số.
- Với i từ n-1 đến 1, thực hiện:
  - Đổi chỗ a[0] với a[i]
  - Dùng thao tác siftDown chuyển i-1 phần tử đầu của mảng thành Heap

```
void siftDown(int arr[], int i, int n)
{
    int maxIndex = i;
    int left = 2*i + 1;
    int right = 2*i + 2;

    if (left < n && arr[left] > arr[maxIndex])
        maxIndex = left;

    if (right < n && arr[right] > arr[maxIndex])
        maxIndex = right;

    if (i != maxIndex)
    {
        swap(arr[i], arr[maxIndex]);
        siftDown(arr, maxIndex, n);
    }
}
```

```
void heapSort(int arr[], int n)
{
    // Xây dựng heap từ mảng
    for (int i = n / 2 - 1; i >= 0; i--)
        siftDown(arr, i, n);

  // Lần lượt đổi phần tử đầu với phần tử
  // thứ i rồi chỉnh lại thành heap
    for (int i = n - 1; i >= 0; i--)
    {
        swap(arr[0], arr[i]);
        siftDown(arr, 0, i);
    }
}
```

Độ phức tạp

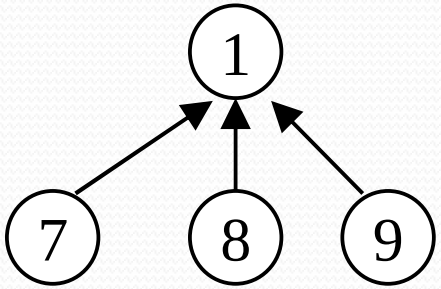
- Thao tác tạo heap từ mảng: thực hiện $n/2$ lần thao tác SiftDown nên có độ phức tạp trong trường hợp xấu nhất là $O(n\log_2 n)$.
- Thực hiện n - 1 lần thao tác SiftDown khi đổi phần tử đầu với phần tử thứ i nên độ phức tạp của thao tác này là $O(n\log_2 n)$.
- Do đó độ phức tạp tính toán của thuật toán HeapSort được đánh giá trong trường hợp xấu nhất là $O(n\log_2 n)$.

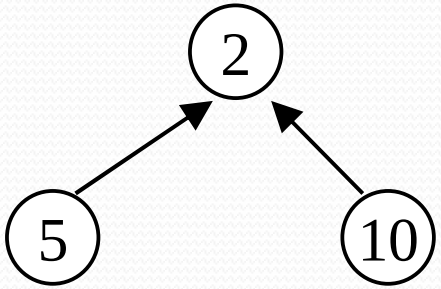
Các tập không giao nhau

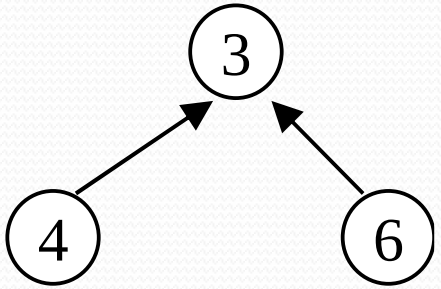
- Disjoin Set là cấu trúc dữ liệu để thao tác với các tập không giao nhau.
- Các thao tác:
  - Union: hợp hai tập hợp thành 1 tập hợp
  - Find: tìm một phần tử thuộc tập hợp nào
- Tổ chức dữ liệu: sử dụng cấu trúc cây, mỗi tập là một cây.
  - Dùng liên kết các nút
  - Dùng mảng

Minh họa

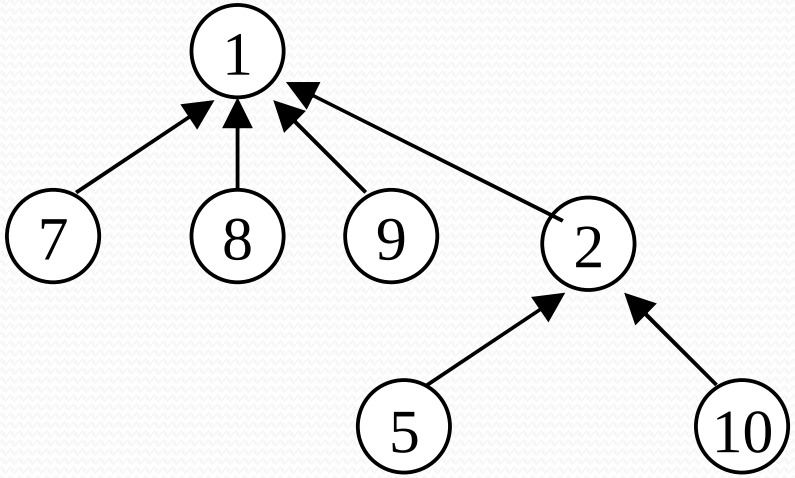
- $S_1 = \{1, 7, 8, 9\}$; $S_2 = \{2, 5, 10\}$; $S_3 = \{3, 4, 6\}$







- Hợp S_1 và S_2



Tổ chức bằng liên kết

```
struct Node {
    int data;
    Node *parent;
    int rank;
};
```

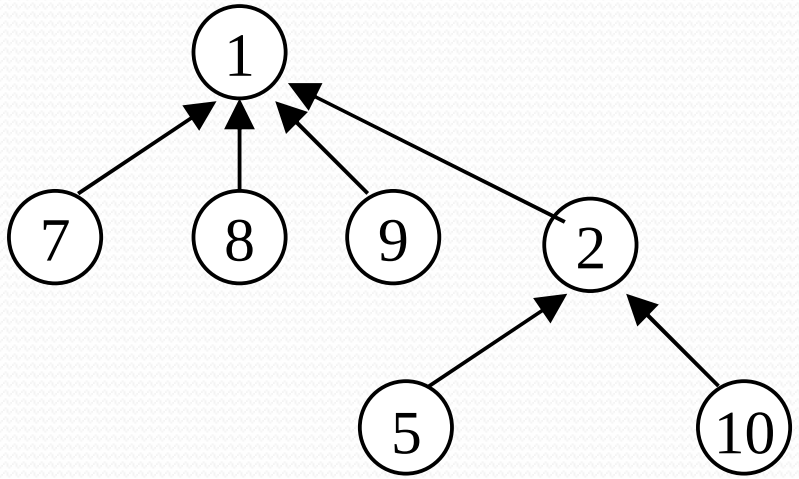
rank là hạng của nút dùng khi hợp hai cây

Tạo tập một nút

```
Node* makeSet(int x) {
    Node *node = new Node;
    node->data = x;
    node->parent = node;
    node->rank = 1;
    return node;
}
```

Tìm nút gốc của một nút

```
Node* findSet(struct Node *node) {
    while (node != node->parent)
        node = node->parent;
    return node->parent;
}
```



- Thuật toán hợp hai cây chứa nút x và y
- Tìm nút gốc của hai tập x và y
- Nếu hai gốc trùng nhau thì hai nút này cùng nằm trong tập hợp
- Ngược lại:
- Nếu gốc của x có rank nhỏ hơn thì gán parent của nó là gốc của y
- Ngược lại nếu gốc của x có rank lớn hơn thì gán parent của y là gốc của x
- Ngược lại thì gán parent của y là gốc của x và tăng rank gốc của x lên 1

```
void unionSet(Node *x, Node *y) {
    Node *xroot = findSet(x);
    Node *yroot = findSet(y);
    if (xroot == yroot) return;
    if (xroot->rank < yroot->rank)
        xroot->parent = yroot;
    else if (xroot->rank > yroot->rank)
        yroot->parent = xroot;
     else {
        yroot->parent = xroot;
        xroot->rank++;
    }
}
```

```
int main() {
    Node *node1 = makeSet(1);
    Node *node2 = makeSet(2);
    Node *node3 = makeSet(3);

    unionSet(node1, node2);
    unionSet(node2, node3);

    Node *root1 = findSet(node1);
    Node *root2 = findSet(node2);
    Node *root3 = findSet(node3);

    cout << "Root of node1 is " << root1->data << endl;
    cout << "Root of node2 is " << root2->data << endl;
    cout << "Root of node3 is " << root3->data << endl;

    return 0;
}
```

Tổ chức dữ liệu bằng mảng

- Dùng mảng parent lưu các nút cha, mảng rank lưu hạng mỗi nút.

```
#define MAX_SIZE 1000
int parent[MAX_SIZE];
int rank[MAX_SIZE];

void makeSet(int x) {
    parent[x] = x;
    rank[x] = 0;
}

int findSet(int x) {
    while (parent[x] != x) {
        x = parent[x];
    }
    return parent[x];
}
```

```
void unionSet(int x, int y) {
    int xroot = findSet(x);
    int yroot = findSet(y);
    if (xroot == yroot) return;
    if (rank[xroot] < rank[yroot]) {
        parent[xroot] = yroot;
    } else if (rank[xroot] > rank[yroot]) {
        parent[yroot] = xroot;
    } else {
        parent[yroot] = xroot;
        rank[xroot]++;
    }
}
```

```
int main() {
    for (int i = 0; i < MAX_SIZE; i++) {
        makeSet(i);
    }

    unionSet(1, 2);
    unionSet(2, 3);

    int root1 = findSet(1);
    int root2 = findSet(2);
    int root3 = findSet(3);

    cout << "Root of 1 is " << root1 << endl;
    cout << "Root of 2 is " << root2 << endl;
    cout << "Root of 3 is " << root3 << endl;

    return 0;
}
```

Áp dụng

- Cho ma trận các số nguyên. Cho biết những ô kề nhau có các số giống nhau.
- Tạo Disjoin Set mà mỗi nút là một ô.
- Duyệt từng ô, nếu ô kề với nó có giá trị going nhau thì hợp hai ô lại.

1	4	1	1
1	1	1	2
1	3	2	2
1	3	3	3
3	2	2	2

```
int main() {
    int matrix[5][4] = {{1, 4, 1, 1},
                        {1, 1, 1, 2},
                        {1, 3, 2, 2},
                        {1, 3, 3, 3},
                        {3, 2, 2, 2}};
    int m = 5, n = 4;

    // Initialize the disjoint sets
    for (int i = 0; i < m * n; i++) {
        make_set(i);
    }
```

```
// Union adjacent cells with the same value
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            int current = i * n + j;

            if (i > 0 && matrix[i][j] == matrix[i - 1][j]) {
                int up = current - n;
                union_sets(current, up);
            }

            if (j > 0 && matrix[i][j] == matrix[i][j - 1]) {
                int left = current - 1;
                union_sets(current, left);
            }
        }
    }
```

```
// Print the disjoint sets
    printf("The sets of adjacent cells with the same value are:\n");
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            int current = i * n + j;

            if (parent[current] == current) {
                cout << "{” ;

                for (int k = 0; k < m * n; k++) {
                    if (parent[k] == current) {
                        cout << k / n << "," << k % n;
                    }
                }

                cout << "}" << endl;
            }
        }
    }
```

Tổng kết

- Heap là một cấu trúc dữ liệu thuận lợi cho các thao tác tìm, lấy ra phần tử có khoá lớn nhất/nhỏ nhất.
- Có thể dùng mảng để lưu trữ heap và thao tác dễ dàng.
- Disjoin Set là cấu trúc dữ liệu để thao tác với các tập không giao nhau.