# Nội dung:

# 1.Các mô hình thuật toán:

## A,Mô hình thuật toán sinh

Thuật toán sinh được dùng để giải lớp các bài toán thỏa mãn hai điều kiện:

•Xác định được một thứ tự trên tập các cấu hình cần liệt kê của bài toán. Biết được cấu hình đầu tiên, biết được cấu hình cuối cùng.

•Từ một cấu hình cuối cùng, ta xây dựng được thuật toán sinh ra cấu hình đứng ngay sau nó theo thứ tự.

## bMô hình thuật toán đệ qui

Thuật toán giải bài toán P thông qua bài toán P’ giống như P được gọi là thuật toán đệ qui. Một hàm được gọi là đệ qui nếu nó được gọi đến chính nó. Một bài toán giải được bằng đệ qui nếu nó thỏa mãn hai điều kiện:

• Phân tích được: Có thể giải được bài toán P bằng bài toán P’ giống như P và chỉ khác P ở dữ liệu đầu vào. Việc giải bài toán P’ cũng được thực hiện theo cách phân tích giống như P.

• Điều kiện dừng: Dãy các bài toán P’ giống như P là hữu hạn và sẽ dừng tại một bài toán xác định nào đó

## bMô hình thuật toán quay lui.

-Quay lui là một kĩ thuật thiết kế giải thuật dựa trên đệ quy. Ý tưởng của quay lui là tìm lời giải từng bước, mỗi bước chọn một trong số các lựa chọn khả dĩ và đệ quy.

-Bản chất của quay lui là một quá trình tìm kiếm theo chiều sâu(Depth-First Search). Về bản chất, tư tưởng của phương pháp là thử từng khả năng cho đến khi tìm thấy lời giải đúng. Đó là một quá trình [tìm kiếm theo độ sâu](https://vi.wikipedia.org/wiki/T%C3%ACm_ki%E1%BA%BFm_theo_chi%E1%BB%81u_s%C3%A2u) trong một tập hợp các lời giải. Trong quá trình tìm kiếm, nếu ta gặp một hướng lựa chọn không thỏa mãn, ta quay lui về điểm lựa chọn nơi có các hướng khác và thử hướng lựa chọn tiếp theo. Khi đã thử hết các lựa chọn xuất phát từ điểm lựa chọn đó, ta quay lại điểm lựa chọn trước đó và thử hướng lựa chọn tiếp theo tại đó. Quá trình tìm kiếm thất bại khi không còn điểm lựa chọn nào nữa.

\*Giả sử ta cần xác định bộ X =(x1, x2,..,xn) thỏa mãn một số ràng buộc nào đó. Ứng với mỗi thành phần xi ta có ni khả năng cần lựa chọn. Ứng với mỗi khả năng j∈ni dành cho thành phần xi ta cần thực hiện:  
• Kiểm tra xem khả năng j có được chấp thuận cho thành phần xi hay không?  
Nếu khả năng j được chấp thuận thì

nếu i là thành phần cuối cùng (i=n) ta ghi nhận nghiệm của bài toán. Nếu i chưa phải cuối cùng ta xác định thành phầnthứ i +1.  
• Nếu không có khả năng j nào được chấp thuận cho thành phần xi thì ta quay lại bước trước đó (i-1) để thử lại các khả năng khác của (𝑖 − 1).

**Nhánh cận :**

Mô hình thuật toán đệ quy quay lui trong chương trước chính là việc tìm kiếm trên cây n cấp sẽ có 2n nút lá điều đó có nghĩa là nếu dữ liệu đầu vào là n thì ta phải duyệt 2n lần trong trường hợp tồi nhất, con số này là quá lớn so với dữ liệu ban đầu và trong quá trình duyệt có một số thao tác thừa đối với việc chọn nghiệm xi tối ưu. Tư tưởng của kĩ thuật nhánh cận là loại bỏ đi những thao tác thừa đó trong quá trình tiến hành gọi thủ tục đệ quy.

Như vậy việc bỏ đi thao tác thừa ở mô hình trên chính là việc thêm vào dòng lệnh **if***(việc thử còn hi vọng tìm ra cấu hình tốt hơn)***then**. Điều này có nghĩa là tại mỗi bước giá trị thử cho xi không còn hi vọng tìm ra cấu hình tốt hơn thì thử giá trị khác ngay mà không cần gọi đệ quy tiếp theo hay ghi nhận cấu hình do vậy nghiệm của bài toán sẽ được làm tốt dần bước sau tối ưu hơn bước cứ như vậy cấu hình cuối cùng sẽ là cấu hình tối ưu nhất.

## cMô hình thuật toán tham lam

-Thuật toán tham lam (Greedy Algorithm): là thuật toán xem xét quá trình giải quyết bài toán bằng việc tạo nên lựa chọn tối ưu cục bộ tại mỗi bước thực hiện với mong muốn tìm ra lựa chọn tối ưu toàn cục. Giải thuật tham lam thường không mang tính tổng quát. Tuy vậy, bằng việc xem xét các lựa chọn tối ưu cục bộ cho ta lời giải gần đúng với phương án tối ưu toàn cục cũng là giải pháp tốt trong thời gian chấp nhận được.

-Để thực hiện một thuật toán tham lam, ta sẽ xác định một cấu trúc con hoặc bài toán con tối ưu trong bài toán đó. Sau đó, xác định giải pháp sẽ bao gồm những gì (ví dụ: tổng lớn nhất, đường đi ngắn nhất, v.v.). Tạo một số cách lặp đi lặp lại để đi qua tất cả các bài toán con và xây dựng giải pháp.-

-Thuật toán tham lam được thực hiện như sau:

* Ban đầu, tập giải pháp (chứa các câu trả lời) là rỗng.
* Ở mỗi bước, một phần tử được thêm vào tập giải pháp.
* Nếu tập giải pháp là khả thi, phần tử hiện tại được giữ lại.
* Mặt khác, phần tử sẽ bị loại bỏ và không bao giờ được xem xét lại.

Nhược điểm

* Khó thiết kế: Khi ta đã tìm ra cách tiếp cận giải thuật tham lam phù hợp, việc thiết kế các thuật toán tham lam có thể dễ dàng. Tuy nhiên, việc tìm ra cách tiếp cận phù hợp có thể sẽ khó.
* Khó xác định: Việc biểu diễn một thuật toán tham lam phù hợp thường đòi hỏi một lập luận sắc bén.

## d,Mô hình thuật toán chia và trị

-Chia để trị là 1 phương pháp áp dụng cho các bài toán có thể giải quyết bằng cách chia nhỏ ra thành các bài toán con từ việc giải quyết các bài toán con này. Sau đó lời giải của các bài toán nhỏ được tổng hợp lại thành lời giải cho bài toán

1. Devide (Chia). Chia bài toán lớn thành những bài toán con có cùng kiểu với bài toán lớn.

2. Conquer (Trị). Giải các bài toán con. Thông thường các bài toán con chỉ khác nhau về dữ liệu vào nên ta có thể thực hiện bằng một thủ tục đệ qui.

3. Combine (Tổng hợp). Tổng hợp lại kết quả của các bài toán con để nhận được kết quả của bài toán lớn

-Vấn đề khó khăn nhất của phương pháp này đó là làm thế nào để chia bài toán lớn một cách hiệu quả thành các bài toán con và dựa trên nguyên tắc nào để phân chia. Nhìn chung quá trình phân chia cần phải đảm bảo hai yếu tố cơ bản đó là: Một là dữ liệu bài toán con phải nhỏ hơn dữ liệu của bài toán ban đầu; Hai là các bài toán con có thể sử dụng cùng một thuật toán hoặc là càng ít thuật toán khác nhau càng tốt điều này sẽ đảm bảo cho việc kết hợp nghiệm các bài toán con thành nghiệm của bài toán ban đầu.

Kĩ thuật nhánh cận và phương pháp chia để trị đã góp phần giải quyết được một số bài toán có dữ liệu đầu vào lớn hoặc những bài toán phức tạp liên quan đến nhiều vấn đề trong thực tiễn góp phần đưa ra được những lời giải *“chấp nhận được”* trên một lĩnh vực nào đó để giải quyết lớp bài toán tối ưu.

## e,Mô hình thuật toán qui hoạch động

Quy hoạch động là một kỹ thuật trong lập trình giúp giải quyết một cách hiệu quả một lớp vấn đề có các bài toán con chồng chéo (gối nhau) và thuộc tính cấu trúc con tối ưu.

* Các bài toán con chồng chéo: Bài toán con là các bài toán nhỏ hơn của bài toán ban đầu. Bất kỳ bài toán nào cũng có các bài toán con trùng nhau nếu việc tìm lời giải của nó liên quan đến việc giải cùng một bài toán con nhiều lần.
* Thuộc tính cấu trúc con tối ưu: Bất kỳ bài toán nào cũng có thuộc tính cấu trúc con tối ưu nếu giải pháp tối ưu tổng thể của nó có thể được xây dựng từ các giải pháp tối ưu của các bài toán con của nó.

-Bài toán lớn cần giải có thể phân rã được thành nhiều bài toán con. Trong đó, sự phối hợp lời giải của các bài toán con cho ta lời giải của bài toán lớn. Bài toán con có lời giải đơn giản được gọi là cơ sở của qui hoạch động. Công thức phối hợp nghiệm của các bài toán con để có nghiệm của bài toán lớn được gọi là công thức truy hồi của qui hoạch động.

- Phải có đủ không gian vật lý lưu trữ lời giải các bài toán con (Bảng phương án của qui hoạch động). Vì qui hoạch động đi giải quyết tất cả các bài toán con, do vậy nếu ta không lưu trữ được lời giải các bài toán con thì không thể phối hợp được lời giải giữa các bài toán con.

- Quá trình giải quyết từ bài toán cơ sở (bài toán con) để tìm ra lời giải bài toán lớn phải được thực hiện sau hữu hạn bước dựa trên bảng phương án của qui hoạch động

**Khi nào thì dùng thuật toán quy hoạch động**

**Bài toán con gối nhau**

Tương tự như thuật toán chia để trị, quy hoạch động cũng chia bài toán lớn thành các bài toán con nhỏ hơn. Quy hoạch động được sử dụng khi các bài toán con này được gọi đi gọi lại. Phương pháp quy hoạch động sẽ lưu kết quả của bài toán con này, và khi được gọi, nó sẽ không cần phải tính lại, do đó làm giảm thời gian tính toán.

Quy hoạch động sẽ không thể áp dụng được (hoặc nói đúng hơn là áp dụng cũng không có tác dụng gì) khi các bài toán con không gối nhau. Ví dụ với thuật toán tìm kiếm nhị phân, quy hoạch động cũng không thể tối ưu được gì cả, bởi vì mỗi khi chia nhỏ bài toán lớn thành các bài toán con, mỗi bài toán cũng chỉ cần giải một lần mà không bao giờ được gọi lại.

**Cấu trúc con tối ưu**

Cấu trúc con tối ưu là một tính chất là lời giải của bài toán lớn sẽ là tập hợp lời giải từ các bài toán nhỏ hơn.Chúng ta sẽ áp dụng phương pháp quy hoạch động bằng cách bắt đầu từ bài toán con dp(0) sau đó tiếp tục với các bài toán con lớn hơn. Lời giải của các bài toán con sẽ được xây dựng lần lượt cho đến chúng ta xây dựng đến bài toán dp(S) và đó chính là kết quả của bài toán lớn. Một điều cần lưu ý với kỹ thuật này là bài toán con tiếp theo sẽ không thể giải được nếu chúng ta chưa giải bài toán con trước đó.

## f,Mô hình thuật toán sắp xếp và tìm kiếm

| **Algorithm** | **Time Complexity** | | |  |
| --- | --- | --- | --- | --- |
|  | **Best** | **Average** | **Worst** |  |
| [Selection Sort](http://geeksquiz.com/selection-sort/) | Ω(n^2) | θ(n^2) | O(n^2) |  |
| [Bubble Sort](http://geeksquiz.com/bubble-sort/) | Ω(n) | θ(n^2) | O(n^2) |  |
| [Insertion Sort](http://geeksquiz.com/insertion-sort/) | Ω(n) | θ(n^2) | O(n^2) |  |
| [Heap Sort](http://geeksquiz.com/heap-sort/) | Ω(n log(n)) | θ(n log(n)) | O(n log(n)) |  |
| [Quick Sort](http://geeksquiz.com/quick-sort/) | Ω(n log(n)) | θ(n log(n)) | O(n^2) |  |
| [Merge Sort](http://geeksquiz.com/merge-sort/) | Ω(n log(n)) | θ(n log(n)) | O(n log(n)) |  |
| [Bucket Sort](https://www.geeksforgeeks.org/bucket-sort-2/) | Ω(n+k) | θ(n+k) | O(n^2) |  |
| [Radix Sort](https://www.geeksforgeeks.org/radix-sort/) | Ω(nk) | θ(nk) | O(nk) |  |

thuật toán tìm kiếm nhị phân

**Chú ý:**Trong bài viết tôi giả sử mảng đang sắp xếp tăng dần. Với trường hợp mảng sắp xếp giảm dần, bạn đọc sau khi hiểu thuật toán sẽ có thể tự làm.

Do tính chất mảng đã sắp xếp, công việc tìm kiếm phần tử x có thể triển khai như sau:

1. Xét đoạn mảng arr[left…right] cần tìm kiếm phần tử x. Ta so sánh x với phần tử ở vị trí giữa của mảng(mid = (left + right)/2). Nếu:
2. Nếu phần tử arr[mid] = x. Kết luận và thoát chương trình.
3. Nếu arr[mid] < x. Chỉ thực hiện tìm kiếm trên đoạn arr[mid+1…right].
4. Nếu arr[mid] > x. Chỉ thực hiện tìm kiếm trên đoạn arr[left…mid-1].

Bằng cách áp dụng thuật toán tìm kiếm nhị phân, độ phức tạp cho trường hợp xấu nhất là O(log(n)).

# 2.Cấu trúc dữ liệu danh sách liên kết

1. **Danh sách liên kết đơn**

* **Định nghĩa:**

Tập hợp các node thông tin (khối dữ liệu) được tổ chức rời rạc trong bộ nhớ. Trong đó, mỗi node gồm hai thành phần:

• Thành phần dữ liệu (infor): dùng để lưu trữ thông tin của node.

• Thành phần con trỏ (pointer): dùng để trỏ đến node dữ liệu tiếp theo.

* **Biểu diễn:** Sử dụng kiểu dữ liệu cấu trúc tự trỏ để định nghĩa mỗi node của danh sách liên kết đơn
* **Các thao tác trên danh sách liên kết đơn**

• Khởi tạo danh sách liên kết đơn: đưa trạng thái danh sách liên kết đơn về trạng thái rỗng. Ta gọi thao tác này là Init().

• Cấp phát miền nhớ cho một node: khi thực hiện thêm node vào danh sách thì node cần thêm vào cần trỏ đến một miền nhớ cụ thể thông qua các thao tác cấp phát bộ nhớ.

• Thêm node vào đầu bên trái danh sách liên kết đơn.

• Thêm node vào đầu bên phải theo chiều con trỏ next.

• Thêm node vào node giữa danh sách liên kết đơn.

• Loại node cuối bên trái danh sách liên kết đơn.

• Loại node cuối bên phải theo chiều con trỏ next.

• Loại node ở giữa danh sách liên kết đơn.

• Duyệt thông tin của danh sách liên kết đơn.

• Tìm node trên danh sách liên kết đơn

* **Ứng dụng của danh sách liên kết**

• Xây dựng các lược đồ quản lý bộ nhớ:

• Thuật toán Best Fit:

• Thuật toán First Fit:

• Thuật toán Best Availbale.

• Biểu diễn ngăn xếp :

• Danh sách L + { Add-Top, Del-Top}.

• Danh sách L + { Add-Bottom, Del-Bottom}.

• Biểu diễn hàng đợi:

• Danh sách L + { Add-Top, Del-Bottom}.

• Danh sách L + { Add-Bottom, Del-Top}.

• Biểu diễn cây.

• Biểu diễn đồ thị

• Biểu diễn tính toán

1. **Danh sách liên kết kép**

* **Định nghĩa.** Tập hợp các node (khối dữ liệu) được tổ chức rời rạc trong bộ nhớ. Trong đó, mỗi node gồm ba thành phần:

• Thành phần dữ liệu (infor): dùng để lưu trữ thông tin của node.

• Thành phần con trỏ prev: dùng để trỏ đến node dữ liệu sau nó.

• Thành phần con trỏ next: dùng để trỏ đến node dữ liệu trước nó.

* **Biểu diễn danh sách liên kết kép:**

typedef struct node {

Item Infor; //Thành phần dữ liệu của node struct node

\*prev; //Thành phần con trỏ sau struct node

\*next; //Thành phần con trỏ trước }\*L;

* **Các thao tác trên danh sách liên kết kép**

• Khởi tạo danh sách liên kết kép.

• Cấp phát miền nhớ cho một node.

• Thêm node vào đầu bên trái danh sách liên kết kép.

• Thêm node vào đầu bên phải danh sách liên kết kép.

• Thêm node vào node giữa danh sách liên kết kép.

• Loại node cuối bên trái danh sách liên kết kép.

• Loại node cuối bên phải danh sách liên kết kép.

• Loại node ở giữa danh sách liên kết kép.

• Duyệt trái danh sách liên kết kép.

• Duyệt phải danh sách liên kết kép.

• Tìm node trên danh sách liên kết kép… Dưới đây là một số thao tác cơ bản trên danh sách liên kết kép.

# 3.Cấu trúc dữ liệu ngăn xếp

## Định nghĩa ngăn xếp

Stack là một kiểu cấu trúc dữ liệu và cơ chế của nó là LIFO (Last In First Out) nghĩa là vào sau ra trước, có thể hình dung Stack như một chồng đĩa, chỉ có thể lấy chiếc đĩa ra hoặc thêm một chiếc đĩa khác vào trên đỉnh của nó và chiếc đĩa nằm trên đỉnh đó được gọi là Top.

## Các phương pháp biểu diễn ngăn xếp

Thực chất **stack** là một**danh sách liên kết**, nên bạn có thể sử dụng một trong các dạng cài đặt sau:

1. Cài đặt stack bằng danh sách liên kết
2. Biểu diễn trực tiếp ngăn xếp bằng mảng một chiều
3. Cài đặt bằng con trỏ (sử dụng các node stack)

Ở bài viết này mình sẽ sử dụng cách thứ 2 tức sử dụng mảng 1 chiều.  
Xem ngăn xếp stack cái đặt bằng con trỏ.

## Các thao tác trên ngăn xếp

* [2.1. Kiểm tra stack đầy(IsFull)](https://nguyenvanhieu.vn/ngan-xep-stack/#21-kiem-tra-stack-dayisfull)
* [2.2. Kiểm tra stack rỗng(IsEmpty)](https://nguyenvanhieu.vn/ngan-xep-stack/#22-kiem-tra-stack-rongisempty)
* [2.3. Thêm phần tử vào đỉnh stack(Push)](https://nguyenvanhieu.vn/ngan-xep-stack/#23-them-phan-tu-vao-dinh-stackpush)
* [2.4. Xóa phần tử khỏi đỉnh stack(Pop)](https://nguyenvanhieu.vn/ngan-xep-stack/#24-xoa-phan-tu-khoi-dinh-stackpop)
* [2.5. Lấy giá trị phần tử ở đỉnh stack(Top)](https://nguyenvanhieu.vn/ngan-xep-stack/#25-lay-gia-tri-phan-tu-o-dinh-stacktop)
* [2.6. Lấy số lượng phần tử stack đang có(Size)](https://nguyenvanhieu.vn/ngan-xep-stack/#26-lay-so-luong-phan-tu-stack-dang-cosize)

## Ứng dụng của ngăn xếp

1. Các ngăn xếp có thể được sử dụng để đánh giá biểu thức.
2. Ngăn xếp có thể được sử dụng để kiểm tra sự khớp dấu ngoặc đơn trong một biểu thức.
3. Ngăn xếp có thể được sử dụng để Chuyển đổi từ dạng biểu thức này sang dạng biểu thức khác.
4. Ngăn xếp có thể được sử dụng để quản lý bộ nhớ.
5. • Biểu diễn tính toán.
6. duyệt đồ thị…

# 4.Cấu trúc dữ liệu hàng đợi

## Định nghĩa hàng đợi

Hàng đợi hay queue là một cấu trúc dữ liệu động hoạt động theo cơ chế FIFO (First In, First Out), nghĩa là phần tử được thêm vào đầu tiên sẽ là phần tử được lấy ra đầu tiên.

Hàng đợi ([tiếng Anh](https://vi.wikipedia.org/wiki/Ti%E1%BA%BFng_Anh): queue) là một [cấu trúc dữ liệu](https://vi.wikipedia.org/wiki/C%E1%BA%A5u_tr%C3%BAc_d%E1%BB%AF_li%E1%BB%87u) dùng để chứa các đối tượng làm việc theo cơ chế FIFO (viết tắt từ tiếng Anh: First In First Out), nghĩa là "vào trước ra trước"

## Các phương pháp biểu diễn hàng đợi

Tương tự như cấu trúc dữ liệu ngăn xếp, thì cấu trúc dữ liệu hàng đợi cũng có thể được triển khai bởi sử dụng

Mảng (Array),

Danh sách liên kết (Linked List),

Con trỏ (Pointer) và Cấu trúc (Struct).

## Các thao tác trên hàng đợi

• Kiểm tra tính rỗng của hàng đợi (Empty(queue q)). Thao tác này được sử dụng khi ta cần đưa dữ liệu vào hàng đợi.

• Kiểm tra tính đầy của hàng đợi (Empty(queue q)). Thao tác này được sử dụng khi ta muốn lấy dữ dữ liệu ra khỏi hàng đợi.

• Thao tác đưa dữ liệu vào hàng đợi (Push(queue q, int x)).

• Thao tác lấy dữ liệu ra khỏi hàng đợi (Pop(queue q))

## Ứng dụng của hàng đợi

Duyệt cây

• Duyệt đồ thị.

• Xây dựng các thuật toán lập lịch.

# 5.Cấu trúc dữ liệu cây

**Một số khái niệm:**

• Node gốc (Root): Node đầu tiên định hình cây.

• Node cha (Father): Node A là node cha của node B nếu B hoặc là node

con bên trái của node A (left son) hoặc B là node con bên phải của node B

(right son).

• Node lá (Leaf): Node không có node con trái, không có node con phải.

• Node trung gian (Internal Node): Node hoặc có node con trái, hoặc có

node con phải, hoặc cả hai hoặc cả hai.

• Node trước (Ancestor): Node A gọi là node trước của node B nếu cây

con node gốc là A chứa node B.

• Node sau trái (left descendent): node B là node sau bên trái của node A

nếu cây con bên trái của node A chứa node B.

• Node sau phải (right descendent): node B là node sau bên phải của

node A nếu cây con bên phải của node A chứa node B.

• Node anh em (brother): A và B là anh em nếu cả A và B là node con trái

và node con phải của cùng một node cha.

• Bậc của node (degree of node): Số cây con tối đa của node.

• Mức của node (level of node): mức node gốc có bậc là 0, mức của các

node khác trên cây bằng mức của node cha cộng thêm 1.

• Chiều sâu của cây (depth of tree): mức lớn nhất của node lá trong cây.

Như vậy, độ sâu của cây bằng đúng độ dài đường đi dài nhất từ node gốc

đến node lá.

## Định nghĩa cây nhị phân/ cây nhị phân tìm kiếm

**Các loại cây nhị phân:**

• Cây lệch trái: Cây chỉ có node con bên trái.

• Cây lệch phải: Cây chỉ có node con bên phải.

•Cây nhị phân đúng (strickly binary tree): Node gốc và tất cả các node trung gian có đúng hai node con.

• Cây nhị phân đầy (complete binary tree): Cây nhị phân đúng và tất cả node lá đều có mức là d.

• Cây nhị phân gần đầy (almost complete binary tree):Cây nhị phân gần đầy cóchiều sâu d là cây nhị phân thỏa mãn: Tất cả node con có mức không nhỏ hơn d-1 đều có hai node con.Các node ở mức d đầy từ trái qua phải.

• Cây nhị phân hoàn toàn cân bằng.Cây nhị phân có số node thuộc nhánh cây con trái và số node thuộc nhánh cây con phải chênh lệch nhau không quá 1.

• Cây nhị phân tìm kiếm. Cây nhị phân thỏa mãn điều kiện:Hoặc là rỗng hoặc có một node gốc. Mỗi node gốc có tối đa hai cây con. Nội dung node gốc lớn hơn nội dungnode con bên trái và nhỏ hơn nội dung node con bên phải. Hai cây con bên trái và bên phải cũng hình thành nên hai cây tìm kiếm.

• Cây nhị phân tìm kiếm hoàn toàn cân bằng. Cây nhị phân tìm kiếm có chiều

sâu cây con bên trái và chiều sâu cây con bên phải chênh lệch nhau không quá 1.

## Biểu diễn cây nhị phân/ cây nhị phân tìm kiếm

Biểu diễn liên tục

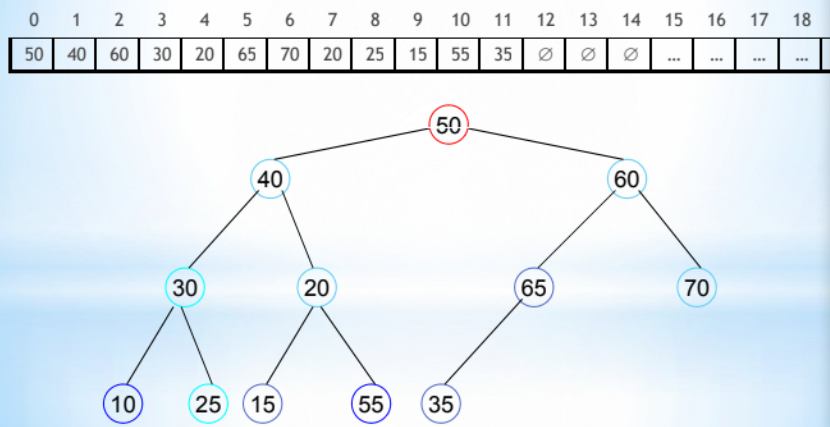
• Biểu diễn liên tục: Sử dụng mảng, trong đó:

• Node gốc: Lưu trữ ở vị trí 0.

• Nếu node cha lưu trữ ở vị trí p thì node con bên trái của nó được lưu trữ ở vị

trí 2p+1, node con phải được lưu trữ ở vị trí 2p+2.

Ví dụ: với cây dưới đây sẽ được lưu trữ trong mảng Tree[MAX] như sau:



4.2.1. Biểu diễn rời rạc:

• Biểu diễn rời rạc: Sử dụng danh sách liên kết.

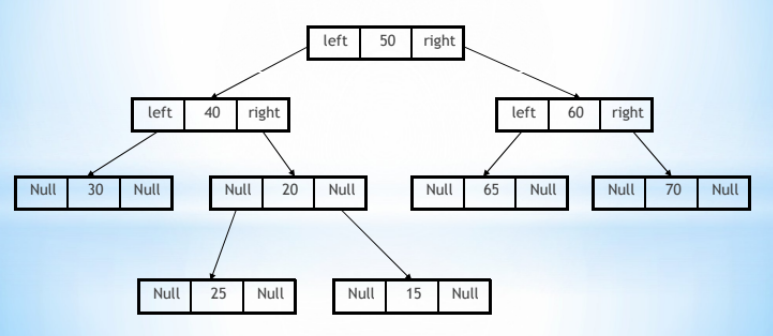
typedef struct node {

Item Infor; //Thông tin của node

struct nde \*left;//Con trỏ node bên trái

struct nde \*right;//Con trỏ node bên phải

} \*Tree;



## Các thao tác trên cây nhị phân/ cây nhị phân tìm kiếm

Các thao tác trên cây nhị phân

• Tạo node gốc cho cây.

• Thêm vào node lá bên trái node p.

• Thêm vào node lá bên phải node p.

• Loại bỏ node lá bên trái node p.

• Loại bỏ node lá bên phải node p.

• Loại bỏ cả cây.

• Tìm kiếm node trên cây.

• Duyệt cây theo thứ tự trước.

• Duyệt cây theo thứ tự giữa.

• Duyệt cây theo thứ tự sau.

Thêm : Kiểm tra tính rỗng của cây T:

4.6. Cây nhị phân tìm kiếm

4.6.1. Định nghĩa

Cây nhị phân thỏa mãn điều kiện:

• Hoặc là rỗng hoặc có một node gốc. Mỗi node gốc có tối đa hai cây con. Nội

dung node gốc lớn hơn nội dung node con bên trái và nhỏ hơn nội dung node

con bên phải.

• Hai cây con bên trái và bên phải cũng hình thành nên hai cây tìm kiếm.

4.6.3. Thao tác

• Tạo node gốc cho cây.

• Thêm vào node vào cây tìm kiếm.

• Loại bỏ node trên cây tìm kiếm.

•Tìm kiếm node trên cây.

• Xoay trái cây tìm kiếm

• Xoay phải cây tìm kiếm

• Duyệt cây theo thứ tự trước.

• Duyệt cây theo thứ tự giữa.

• Duyệt cây theo thứ tự sau.

4.7. Cây nhị phân tìm kiếm cân bằng

• Định nghĩa. Là cây nhị phân tìm kiếm tự cân bằng. Cây tìm kiếm cân bằng có

tính chất độ cao của cây con bên trái và độ cao cây con bên phải chên lệch nhau

không quá 1.

• Cây AVL: Cho phép ta tìm kiếm, bổ sung, loại bỏ node nhanh hơn cây nhị phân

thông thường vì độ cao của cây luôn là O(log (n)).

4.8. Cây nhiều nhánh: Tập hữu hạn các node có cùng kiểu dữ liệu được chia

thành các tập, trong đó:

• Hoặc là tập rỗng hoặc có một node gốc, mỗi node gốc có thể có nhiều khóa đã

được sắp xếp.

• Các tập con còn lại hoặc là rỗng hoăc cũng tự hình thành nên các cây cây nhiều

nhánh.

## Ứng dụng của cây nhị phân/ cây nhị phân tìm kiếm

4. 4. Ứng dụng của cây nhị phân

4.1.4.1. Kích cỡ cây nhị phân (size of a tree). Ta định nghĩa kích cỡ của một cây là số node có thực trên cây.

4.4. 2. Xác định hai cây nhị phân giống nhau. Ta định nghĩa hai cây nhị phân giống nhau nếu chúng có cùng chung node và mỗi node được sắp đặt giống nhau trên cây.

4.4.3. Tìm độ cao của cây. Độ cao của cây được định nghĩa là đường đi dài nhất từ node gốc đến node lá.

4.4.4. Cây phản chiếu (Mirror Tree). Cây phản chiếu của cây nhị phân T là cây nhị phân M(T), trong đó cây con bên trái của cây T trở thành cây con bên phải của M(T) và cây con bên phải của T trở thành cây con trái của M(T). Bài toán được đặt ra l hãy dịch chuyển cây nhị phân T cho trước thành cây nhị phân phản chiểu của T là cây M(T).

4.4.5. Tìm độ cao của cây. Độ cao của cây được định nghĩa là đường đi dài nhất từ node gốc đến node lá.

4.4.6. Tìm tất cả đường đi từ node gốc đến node lá

4.4.7. Đến tất cả các node lá trên cây

4.4.8. Kiểm tra một cây thỏa mãn điều kiện node trung gian bằng tổng hai node con trái và phải hay không? Node không có node lá trái hoặc phải được xem có giá trị 0.

## Common Data Structure Operations

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Data Structure | Time Complexity | | | | | | | | Space Complexity |
|  | Average | | | | Worst | | | | Worst |
|  | Access | Search | Insertion | Deletion | Access | Search | Insertion | Deletion |  |
| [Array](http://en.wikipedia.org/wiki/Array_data_structure) | Θ(1) | Θ(n) | Θ(n) | Θ(n) | O(1) | O(n) | O(n) | O(n) | O(n) |
| [Stack](http://en.wikipedia.org/wiki/Stack_(abstract_data_type)) | Θ(n) | Θ(n) | Θ(1) | Θ(1) | O(n) | O(n) | O(1) | O(1) | O(n) |
| [Queue](http://en.wikipedia.org/wiki/Queue_(abstract_data_type)) | Θ(n) | Θ(n) | Θ(1) | Θ(1) | O(n) | O(n) | O(1) | O(1) | O(n) |
| [Singly-Linked List](http://en.wikipedia.org/wiki/Singly_linked_list#Singly_linked_lists) | Θ(n) | Θ(n) | Θ(1) | Θ(1) | O(n) | O(n) | O(1) | O(1) | O(n) |
| [Doubly-Linked List](http://en.wikipedia.org/wiki/Doubly_linked_list) | Θ(n) | Θ(n) | Θ(1) | Θ(1) | O(n) | O(n) | O(1) | O(1) | O(n) |
| [Skip List](http://en.wikipedia.org/wiki/Skip_list) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(n) | O(n) | O(n) | O(n) | O(n log(n)) |
| [Hash Table](http://en.wikipedia.org/wiki/Hash_table) | N/A | Θ(1) | Θ(1) | Θ(1) | N/A | O(n) | O(n) | O(n) | O(n) |
| [Binary Search Tree](http://en.wikipedia.org/wiki/Binary_search_tree) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(n) | O(n) | O(n) | O(n) | O(n) |
| [Cartesian Tree](https://en.wikipedia.org/wiki/Cartesian_tree) | N/A | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | N/A | O(n) | O(n) | O(n) | O(n) |
| [B-Tree](http://en.wikipedia.org/wiki/B_tree) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n) |
| [Red-Black Tree](http://en.wikipedia.org/wiki/Red-black_tree) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n) |
| [Splay Tree](https://en.wikipedia.org/wiki/Splay_tree) | N/A | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | N/A | O(log(n)) | O(log(n)) | O(log(n)) | O(n) |
| [AVL Tree](http://en.wikipedia.org/wiki/AVL_tree) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(log(n)) | O(n) |
| [KD Tree](http://en.wikipedia.org/wiki/K-d_tree) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | Θ(log(n)) | O(n) | O(n) | O(n) | O(n) | O(n) |

## Array Sorting Algorithms

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algorithm | Time Complexity | | | Space Complexity |
|  | Best | Average | Worst | Worst |
| [Quicksort](http://en.wikipedia.org/wiki/Quicksort) | Ω(n log(n)) | Θ(n log(n)) | O(n^2) | O(log(n)) |
| [Mergesort](http://en.wikipedia.org/wiki/Merge_sort) | Ω(n log(n)) | Θ(n log(n)) | O(n log(n)) | O(n) |
| [Timsort](http://en.wikipedia.org/wiki/Timsort) | Ω(n) | Θ(n log(n)) | O(n log(n)) | O(n) |
| [Heapsort](http://en.wikipedia.org/wiki/Heapsort) | Ω(n log(n)) | Θ(n log(n)) | O(n log(n)) | O(1) |
| [Bubble Sort](http://en.wikipedia.org/wiki/Bubble_sort) | Ω(n) | Θ(n^2) | O(n^2) | O(1) |
| [Insertion Sort](http://en.wikipedia.org/wiki/Insertion_sort) | Ω(n) | Θ(n^2) | O(n^2) | O(1) |
| [Selection Sort](http://en.wikipedia.org/wiki/Selection_sort) | Ω(n^2) | Θ(n^2) | O(n^2) | O(1) |
| [Tree Sort](https://en.wikipedia.org/wiki/Tree_sort) | Ω(n log(n)) | Θ(n log(n)) | O(n^2) | O(n) |
| [Shell Sort](http://en.wikipedia.org/wiki/Shellsort) | Ω(n log(n)) | Θ(n(log(n))^2) | O(n(log(n))^2) | O(1) |
| [Bucket Sort](http://en.wikipedia.org/wiki/Bucket_sort) | Ω(n+k) | Θ(n+k) | O(n^2) | O(n) |
| [Radix Sort](http://en.wikipedia.org/wiki/Radix_sort) | Ω(nk) | Θ(nk) | O(nk) | O(n+k) |
| [Counting Sort](https://en.wikipedia.org/wiki/Counting_sort) | Ω(n+k) | Θ(n+k) | O(n+k) | O(k) |
| [Cubesort](https://en.wikipedia.org/wiki/Cubesort) | Ω(n) | Θ(n log(n)) | O(n log(n)) | O(n) |

1. Đồ thị
   1. Định nghĩa và khái niệm trên đồ thị vô hướng/có hướng.
   2. Các phương pháp biểu diễn trên đồ thị vô hướng/có hướng.
   3. Thuật toán DFS/BFS trên đồ thị vô hướng/có hướng.
   4. Các thuật toán xây dựng cây khung của đồ thị
   5. Các thuật toán xây tìm chu trình/ đường đi Euler trên đồ thị.
   6. Các thuật toán tìm đường đi ngắn nhất trên đồ thị.

# 6.Đồ thị

1. **Đồ thị vô hướng**. Đồ thị G= được gọi là đồ thị vô hướng nếu các cạnh thuộc E là các bộ không tính đến thứ tự các đỉnh trong V

Đơn đồ thị có hướng. Đồ thị G = bao gồm V là tập các đỉnh, E là tập các cặp có thứ tự gồm hai phần tử của V gọi là các cung. Giữa hai đỉnh bất kỳ của G tồn tại nhiều nhất một cung

1. **PHương pháp biểu diễn**

**1.1 Biểu diễn ma trận kề:**

• Dòng đầu tiên ghi lại số tự nhiên N là số đỉnh của đồ thị;

• N dòng kế tiếp ghi lại ma trận kề của đồ thị. Hai phần tử khác nhau của ma trận kề được ghi cách nhau một vài khoảng trống.

**1.2 Biểu diễn bằng danh sách cạnh:**

• Dòng đầu tiên ghi lại hai số tự nhiên N và M tương ứng với là số đỉnh và số cạnh của đồ thị. Hai số được viết cách nhau một vài khoảng trống.

• M dòng kế tiếp, mỗi dòng ghi lại một cạnh của đồ thị. Đỉnh đầu và đỉnh cuối của mỗi cạnh được viết cách nhau một vài khoảng trống.

**1.3 Biểu diễn đồ thị bằng danh sách kề:**

• Dòng đầu tiên ghi lại số đỉnh của đồ thị.

• Những dòng kế tiếp ghi lại trên mỗi dòng danh sách kề của đỉnh tương ứng. Danh sách kề của mỗi đỉnh được ghi theo khuôn dạng: + < danh sách đỉnh kề của đỉnh>. Các đỉnh (số) được viết cách nhau một vài khoảng trống.

1. DFS
2. BFS
3. **Cây khung đồ thị**
4. **Thuật toán TreeDfs(u)**:

2. kruskal

3. prim

1. **Thuật toán: *Tìm chu trình/đường đi Euler***

distra

bellman ford