BÁO CÁO MÔ TẢ THUẬT TOÁN DSA – TUẦN 4

THUẬT TOÁN SINGLY LINKED LIST VÀ DOUBLY LINKED LIST

Nguyễn Đình Trung Kiên – 24120195

I. SINGLY LINKED LIST:

1. createNode:

- Với một số nguyên được đưa vào hàm, ta cần trả về con trỏ đến một NODE được tạo mới chứa dữ liệu là số nguyên đó.
- Ta tạo mới một NODE, nếu tạo được (bộ nhớ có thể cấp phát) thì đưa số nguyên được cung cấp vào lưu trữ của NODE đó, đồng thời cho NODE chỉ về trỏ NULL.
- Trả về con trỏ đến NODE vừa tạo.

2. createList:

- Với con trỏ tới một NODE nào đó được đưa vào hàm, ta trả về địa chỉ tới một LIST được tạo mới và chỉ chứa duy nhất NODE đó.
- Ta tạo mới một LIST, nếu tạo được thì ta trỏ cả Head và Tail của LIST đó vào NODE được cung cấp, vì chỉ có duy nhất một NODE nên NODE đó vừa là Tail, vừa là Head.
- Trả về con trỏ đến LIST vừa tạo.

3. addHead:

- Với một LIST cùng một số nguyên được đưa vào hàm, ta cần thêm NODE chứa số nguyên đó vào đầu LIST và cập nhật lại Head.
- Nếu LIST không có NODE nào, ta đưa NODE được tạo bởi số nguyên được cung cấp vào LIST, trỏ cả Head và Tail vào NODE đó vì đó là NODE duy nhất của LIST.
- Nếu LIST đã có các NODE như bình thường, ta nối NODE được tạo vào đầu LIST rồi cập nhật lại Head cho LIST.

- Trả về true nếu đã thực hiện thành công.

4. addTail:

- Với một LIST cùng một số nguyên được đưa vào hàm, ta cần thêm NODE chứa số nguyên đó vào cuối LIST và cập nhật lại Tail.
- Nếu LIST không có NODE nào, ta thực hiện như hàm **addHead**. Vì khi đó Head và Tail của LIST là cùng một NODE.
- Nếu LIST đã có các NODE như bình thường, ta nối NODE được tạo vào cuối LIST rồi cập nhật lại Tail cho LIST.
- Trả về true nếu thực hiện thành công.

5. removeHead:

- Với một LIST được đưa vào hàm, ta xóa đi NODE đầu tiên và cập nhật lại Head cho LIST đó.
- Nếu LIST không có phần tử nào, ta kết thúc hàm, trả về false.
- Ngược lại, ta cần đưa cập nhật lại Head đến NODE tiếp theo sau Head cũ, đồng thời giữ lại phần tử Head ban đầu để xóa đi.
- Tuy nhiên, nếu ở trường hợp LIST chỉ có duy nhất một NODE và sau khi xóa, LIST không còn NODE nào thì ta cần cập nhật lại cả Head và Tail trỏ về NULL.
- Trả về true nếu thực thi thành công các thao tác.

6. removeTail:

- Tương tự như hàm **removeHead** nhưng thay vì cần xóa đi NODE đầu tiên, ta phải xóa đi NODE cuối cùng.
- Thay vì có thể truy xuất đến ngay như NODE Head, để đến được NODE liền trước Tail ta cần duyệt đến cuối LIST.
- Dừng lại ở phần tử liền kề trước NODE Tail, ta giữ lại NODE Tail ban đầu, cập nhật lại NODE Tail và xóa NODE Tail ban đầu đi. Đồng thời cần cho NODE Tail mới trỏ đến NULL để kết thúc LIST đúng cách.

7. removeAll:

- Ta cần xóa hết tất cả các NODE của LIST được đưa vào hàm.
- Nếu LIST rỗng, ta kết thúc hàm vì không có gì để xóa.
- Lần lượt xóa Head của LIST rồi lại cập nhật lại Head cho đến khi LIST rỗng.

8. removeBefore:

- Với LIST được đưa vào cùng một số nguyên Val, ta cần xóa đi NODE đằng trước của NODE có giá trị Val trong LIST.
- Ta sử dụng 2 con trỏ để tìm đến NODE cần xóa. Một con trỏ để tìm NODE cần xóa (NODE có NODE kế nó là giá trị Val) và một con trỏ theo ngay sau NODE trên để xóa NODE khi tìm thấy (Vì khi xóa ta cần có 2 NODE liền sau và trước NODE cần xóa để nối lại sau khi loại NODE đó ra khỏi LIST).
- Đưa 2 con trỏ vào vòng lặp, tìm đến khi đã thấy NODE cần tìm hoặc duyệt đến hết LIST (đến NODE Tail vì NODE Tail không thể là NODE liền trước của NODE nào).
- Nếu tìm thấy NODE (chưa duyệt đến NODE Tail), ta thực hiện việc xóa NODE đó đi thông qua con trỏ phụ liền trước (Tuy nhiên cần kiểm tra xem NODE đó có phải NODE Head hay không để có cách xóa hợp lý).

9. removeAfter:

- Tương tự như hàm **removeBefore,** nhưng ta lại cần xóa đi NODE ngay sau NODE có giá trị Val.
- Ở hàm này ta chỉ cần sử dụng 1 con trỏ là có thể thực thi xóa NODE ngay sau nó.
- Đến khi tìm thấy NODE có giá trị Val, ta lại cần kiểm tra xem NODE ngay sau nó có tồn tại để xóa hay không.
- Nếu đã xóa được, ta lại cần kiểm tra xem NODE cần xóa có phải là Tail của LIST hay không, nếu có, ta sử dụng lại làm **removeTail** để có cách xóa hợp lý.

10. addPos:

- Từ một LIST được đưa vào, ta cần thêm NODE mới có giá trị data vào vị trí pos.
- Sử dụng 2 con trỏ để thực hiện tìm kiếm NODE tại vị trí pos và chèn vào vị trí đó (một con trỏ để tìm tới pos và một con trỏ theo sau để thực hiện chèn).
- Ở trường hợp đã duyệt tới vị trí pos và tìm ra NODE hợp lệ ở đó, ta cần kiểm tra xem đó có phải là Head hay không để chèn vào Head một cách hợp lệ, nếu không, ta thực hiện chèn thông qua con trỏ phụ liền trước như bình thường.
- Nếu không tìm thấy NODE hợp lệ tại pos, có 2 trường hợp xảy ra:
  - pos quá lớn so với LIST, vị trí chèn không hợp lệ. Ta không cần làm gì thêm và trả về false.
  - pos đã được giảm về 0, nhưng con trỏ tìm tới một vị trí NULL, thì đó có thể là ta cần chèn vào Tail cho LIST hoặc LIST rỗng và ta cần chèn một NODE vào LIST đó. Cả hai trường hợp trên đều có thể giải quyết bằng hàm **addTail** vì hàm đã có sẵn trường hợp LIST rỗng.

11. removePos:

- Trái lại với hàm trên, ở hàm này ta lại phải xóa đi NODE ở vị trí pos.
- Sử dụng 2 con trỏ như hàm **removeBefore** để tìm tới NODE cần xóa và xóa NODE đó.
- Nếu đã tìm thấy NODE cần xóa (tìm thấy một NODE hợp lệ tại vị trí pos), ta kiểm tra 2 trường hợp đặc biệt là có phải xóa Head hay Tail hay không để cập nhật đúng cách. Nếu không, ta sử dụng 2 con trỏ trên để xóa như bình thường.

12. addBefore:

- Từ LIST được cung cấp, ta chèn NODE mới lưu trữ số nguyên data vào trước NODE có giá trị val.
- Nếu LIST rỗng, ta kết thúc hàm vì không có NODE nào để xác định vị trí trước nó.
- Ta dùng 2 con trỏ để tìm ra vị trí cần chèn trước và thực hiện thao tác chèn (1 con trỏ để tìm NODE có data là Val và một con trỏ liền sau để chèn trước NODE đó).
- Nếu tìm ra vị trí cần chèn trước, ta thực hiện chèn với con trỏ phụ như bình thường nhưng cần kiểm tra xem NODE đó có phải là Head hay không.
- Trả về true nếu thực hiện được thao tác chèn trên, ngược lại trả về false.

13. addAfter:

- Ngược lại với hàm trước, ta phải chèn vào LIST được cung cấp NODE mới lưu trữ số nguyên data ngay sau NODE có giá trị Val.
- Vì là chèn sau NODE có giá trị Val, ta không cần lưu NODE liền trước (sử dụng 2 con trỏ liền nhau) mà chỉ cần tìm tới NODE đó bằng chính con trỏ tới NODE đó, vì ta có thể nối ngay từ NODE tìm được.
- Nếu tìm thấy NODE thỏa yêu cầu trong LIST, ta kiểm tra xem có cần chèn vào Tail hay không (không xét Head vì Head không thể là NODE sau của bất kì NODE nào), nếu không thì ta chèn NODE mới như bình thường. Thực hiện thành công thì trả về true.
- Nếu không, ta trả về false.

14. printList:

- Ta cần in ra giá trị mà các NODE trong LIST lưu trữ.
- Duyệt cả LIST đến khi đến NULL (kết thúc LIST) và in ra từng giá trị của NODE duyệt tới.

15. countElements:

- Ta cần tìm ra số lượng NODE trong LIST được cung cấp.
- Ta dùng một biến đếm và duyệt cả LIST để đếm rồi trả về biến đếm đó.

16. reverseList:

- Ta cần trả về con trỏ đến một LIST mới là LIST đảo ngược của LIST được cung cấp.
- Nếu LIST rỗng hoặc chỉ có một phần tử thì đương nhiên không cần đảo ngược.
- Ở phần Test Cases, ta phải giải phóng bộ nhớ cả LIST ban đầu lẫn LIST đảo ngược nên ta không thể sử dụng thuật toán đảo ngược ngay trên LIST ban đầu đó mà phải tạo LIST mới.
- Ta tạo một LIST mới rồi thêm mới các NODE theo như LIST cũ. Để tạo một LIST ngược, ta duyệt xuôi LIST cũ rồi chèn ngược vào LIST mới (LIST cũ ta duyệt từ đầu tới cuối nên LIST mới ta phải chèn từ cuối tới đầu, nghĩa là liên tục chèn vào Head).
- Sau khi hoàn thành, ta trả về con trỏ đến LIST đã đảo ngược.

17. removeDuplicate:

- Ta cần loại bỏ tất cả NODE có giá trị data trùng nhau trong LIST, đối với các NODE như thế, ta chỉ giữ lại một NODE trong LIST.
- Ta dùng một con trỏ để duyệt qua các NODE trong mảng (chỉ duyệt tới Tail, hoặc duyệt tới khi NODE đó còn hợp lệ vì có thể thao tác xóa sẽ xóa hết các NODE sau nó). Dùng NODE này để đánh dấu lại giá trị cần kiểm tra, sau đó sử dụng 2 con trỏ để tìm tới các NODE trùng giá trị với NODE được đánh dấu cũng như thực hiện thao tác xóa NODE.
- Khi đã tìm thấy NODE cần xóa ở con trỏ chính, trước hết ta lưu lại NODE liền sau để duyệt tiếp (vì sau khi xóa NODE ở vị trí tìm được ta không thể truy xuất tiếp tới NODE tiếp theo).

- Thực hiện kiểm tra xem NODE đó có phải là NODE Tail hay không, nếu có thực hiện xóa Tail, nếu không ta xóa NODE đó như bình thường với 2 con trỏ.
- Với thao tác xóa NODE, ta chỉ cần cập nhật lại NODE chính tới NODE tiếp theo đã được đánh dấu lại để duyệt tiếp, không cần cập nhật NODE phụ liền trước vì khi đó NODE phụ vẫn là NODE liền trước của NODE chính.

18. removeElement:

- Ta cần xóa hết tất cả các NODE có giá trị key trong LIST được cung cấp.
- Ta sử dụng 2 con trỏ để tìm và thực hiện xóa NODE (một con trỏ để tìm và một con trỏ theo ngay sau con trỏ đó để thực hiện thao tác xóa).
- Duyệt 2 con trỏ qua từng NODE trong LIST. Nếu tìm thấy NODE cần xóa, ta cần kiểm tra xem NODE đó có là Tail hoặc Head hay không, nếu không thì ta thực hiện xóa thông qua con trỏ phụ như bình thường. Tuy nhiên, ở mỗi lần duyệt ta cần lưu lại con trỏ liền sau để thực hiện bước nhảy, vì nếu cần xóa NODE nào đó, ta không thể truy xuất đến NODE liền sau được mà phải dựa vào việc lưu lại NODE liền sau đó trong mỗi lần duyệt.
- Cách duyệt theo bước nhảy của 2 thao tác xóa và không xóa cũng khác nhau. Nếu cần xóa NODE đó, sau khi xóa xong thì ta chỉ cần cập nhật con trỏ chính để tìm tiếp vì rõ ràng khi đó NODE phụ không cần đổi vị trí (nó vẫn là NODE liền trước của NODE chính). Còn nếu không cần xóa, ta cho 2 NODE thực hiện bước nhảy như bình thường.
- Trả về true nếu thực hiện bình thường.

II. DOUBLY LINKED LIST:

1. createNode:

- Thao tác tương tự như Singly Linked List, tuy nhiên ta còn phải cho NODE trỏ cả trước và sau vào NULL.

2. *createList:*

- Thao tác tương tự Singly Linked List.

3. *addHead:*

- Thao tác tương tự Singly Linked List.
- Tuy nhiên, khi ta nối Head mới vào LIST, ta phải nối cả trỏ Next của Head mới vào Head cũ và cả trỏ Prev của Head cũ vào Head mới rồi cập nhật lại Head.

4. *addTail:*

- Thao tác tương tự như Singly Linked List.
- Tuy nhiên cũng tương tự như hàm **addHead** trên, ta phải nối trỏ Next của Tail cũ đến Tail mới rồi trỏ Prev của Tail mới tới Tail cũ, sau đó cập nhật lại Tail thì mới hoàn thành.

5. *removeHead:*

- Thao tác tương tự như Singly Linked List.
- Tuy nhiên, ngoài việc cập nhật lại Head, ta còn cần cho trỏ Prev của Head mới về NULL.

6. *removeTail:*

- Thao tác tương tự như Singly Linked List.
- Tuy nhiên ta có thể truy xuất ngược lại NODE liền trước của Tail cũ để cập nhật Tail mới nhờ vào trỏ Prev chứ không cần duyệt tuyến tính như ở Singly Linked List nữa.

7. *removeAll:*

- Thao tác tương tự như Singly Linked List.

8. *removeBefore:*

- Thao tác gần như tương tự Singly Linked List.

- Thay vì phải sử dụng 2 con trỏ để tìm và hỗ trợ việc xóa NODE, ở DLL ta chỉ cần sử dụng 1 con trỏ để duyệt và tìm NODE, vì ta có thể truy xuất cả NODE trước và sau NODE tìm tới.
- Khi đã tìm tới NODE chứa giá trị cần tìm, ta cần kiểm tra xem NODE liền trước đó có hợp lệ hay không mới tiến hành xóa (bởi nếu NODE duyệt tới là NODE Head thì không có NODE liền trước để xóa).
- Khi xóa cũng cần kiểm tra xem NODE liền trước NODE đã duyệt tới (NODE cần xóa) có phải là Head hay không để tiến hành xóa hợp lệ, nếu không thì ta xóa như bình thường, sử dụng linh hoạt 2 con trỏ Prev và Next để xóa.

9. removeAfter:

- Thao tác tương tự như **removeBefore** phía trên.
- Tuy nhiên, thay vì xóa NODE liền trước, ta phải xóa NODE liền sau.
- Trường hợp đặc biệt thay vì kiểm tra NODE cần xóa có phải là Head, ta lại cần kiểm tra NODE cần xóa có phải là Tail hay không.

10. addPos:

- Thao tác gần như tương tự như với Singly Linked List.
- Tuy nhiên, ta cũng không cần dùng 2 con trỏ để tìm vị trí pos cần chèn mà chỉ cần dùng 1 con trỏ đưa vào vòng lặp cùng pos, giảm pos đến khi tìm ra vị trí chèn (pos giảm về 0) hoặc khi không thể duyệt tiếp trong LIST được nữa (con trỏ duyệt đã trỏ tới NULL).
- Việc chèn NODE mới cũng cần thêm bước nối trỏ Prev vào các NODE cần nối chứ không chỉ trỏ Next như SLL.

11. removePos:

- Thao tác gần như tương tự với Singly Linked List.

- Tuy nhiên, ta không cần sử dụng 2 con trỏ để tìm và xóa NODE mà chỉ cần sử dụng 1 con trỏ vì ta có thể truy xuất tới NODE liền trước.

12. addBefore:

- Thao tác gần như tương tự với Singly Linked List.
- Tương tự như các thuật toán trên, ta cũng chỉ cần dùng 1 con trỏ duy nhất để duyệt các NODE và thao tác thêm NODE cũng sẽ nhiều bước hơn đối với Singly Linked List, vì ta phải nối cả trỏ Prev của các NODE vào với NODE cần nối để hoàn thành việc chèn.

13. addAfter:

- Gần như tương tự với Singly Linked List, chỉ khác biệt ở thao tác chèn NODE như mô tả của hàm **addBefore** phía trên.

14. printList:

- Thao tác tương tự như Singly Linked List.

15. countElements:

- Thao tác tương tự như Singly Linked List.

16. reverseList:

- Thao tác tương tự như Singly Linked List.

17. removeDuplicate:

- Thao tác tương tự như Singly Linked List.
- Ta chỉ cần dùng một con trỏ để duyệt tìm thay vì phải sử dụng tới hai con trỏ.

18. removeElement:

- Thao tác tương tự như Singly Linked List.
- Ta cũng chỉ cần dùng một con trỏ để duyệt tìm.

MINH CHỨNG SOURCE CODE ĐÃ ĐƯỢC PUSH LÊN GITHUB: