ĐẠI HỌC HUẾ
KHOA KỸ THUẬT VÀ CÔNG NGHỆ

BÁO CÁO ĐỒ ÁN
HỌC KÌ I, năm học 2022-2023

Học phần:
CẤU TRÚC DỮ LIỆU & GIẢI THUẬT

Số phách **(**Do hội đồng chấm ghi thi)

Thừa Thiên Huế, tháng 04 năm 2023.

ĐẠI HỌC HUẾ
KHOA KỸ THUẬT VÀ CÔNG NGHỆ

BÁO CÁO ĐỒ ÁN
HỌC KÌ I, năm học 2022-2023

Học phần:
CẤU TRÚC DỮ LIỆU & GIẢI THUẬT

Giảng viên hướng dẫn: Nguyễn Thanh Nam.
Lớp: Khoa học dữ liệu và Trí tuệ nhân tạo khóa 3.
Sinh viên thực hiện: Phạm Phước Bảo Tín_22E1020021.
(ký và ghi rõ họ tên)

Số phách **(**Do hội đồng chấm ghi thi)

Thừa Thiên Huế, tháng 04 năm 2023

ĐẠI HỌC HUẾ
KHOA KỸ THUẬT VÀ CÔNG NGHỆ

PHIẾU ĐÁNH GIÁ ĐỒ ÁN/TIỂU LUẬN/BÀI TẬP LỚN
Học kỳ II, năm học 2022 - 2023

Cán bộ chấm thi 1	**Cán bộ chấm thi 2**
Nhận xét: .... **Điểm đánh giá của CBCT1:** Bằng số: ..... Bằng chữ:	**Nhận xét:** .... **Điểm đánh giá của CBCT2:** Bằng số: ..... Bằng chữ: .....

Điểm kết luận: .....

Bằng số:.....

Bằng chữ: .....

Thừa Thiên Huế, ngày 07 tháng 06 năm 2023

Cán bộ chấm thi 1	**Cán bộ chấm thi 2**
(Ký và ghi rõ họ và tên)	*(Ký và ghi rõ họ và tên)*

LỜI CẢM ƠN

Được trở thành sinh viên Khoa Kỹ Thuật và Công Nghệ - Đại học Huế em rất hạnh phúc và biết ơn. Hạnh phúc vì mình đã đạt được mục tiêu mong muốn và biết ơn sự cống hiến, chỉ bảo tận tình sâu sắc của quý thầy cô trong khoa đồng thời đã tạo điều kiện học tập lí tưởng cho chúng em. Để hoàn thành đồ án một cách chỉnh chu nhất có thể em xin gửi lời cảm ơn đến thầy giáo bộ môn – Thầy Nguyễn Thanh Nam đã hướng dẫn tận tình, chi tiết cho chúng em trong quá trình hoàn thành đồ án lẫn quá trình học tập học của sinh viên chúng em. Hi vọng rằng thời gian sắp tới em sẽ luôn cố gắng, nổ lực hơn nữa trong học tập chuyên nghành của mình.

Trong quá trình hoàn thành đồ án mặc dù em đã chuẩn bị kĩ nhưng không thể tránh khỏi những sai sót, em mong nhận được sự góp ý từ quý thầy, cô. Lời cuối cùng em xin kính chúc quý thầy, cô thật nhiều sức khỏe để tiếp tục dẫn dắt chúng em và những thế hệ tiếp theo thành người.

DANH MỤC HÌNH ẢNH

DANH MỤC BẢNG BIỂU

MỤC LỤC

CHƯƠNG 1: ĐỆ QUY

1.1 Tháp Hà Nội

1.1.1 Giới thiệu về bài toán Tháp Hà Nội

Bài toán Tháp Hà Nội là một trò chơi toán học. Yêu cầu của trò chơi là di chuyển toàn bộ số đĩa sang một cọc khác, tuân theo các quy tắc sau:

- Chỉ có 3 cột để di chuyển.
- Một lần chỉ được di chuyển một đĩa (không được di chuyển đĩa nằm giữa).
- Một đĩa chỉ có thể đặt lên một đĩa lớn hơn nó.

Để giải bài toán này chúng ta có thể sử dụng giải thuật đệ quy.

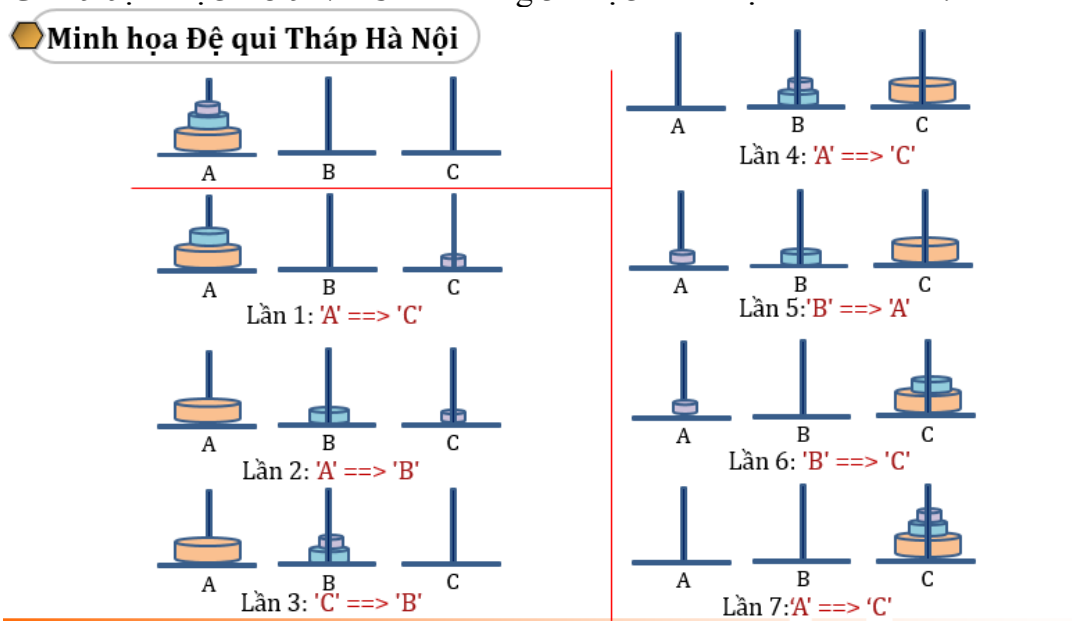
1.1.2 Giải thuật đệ quy trong bài toán Tháp Hà Nội

Đặt tên cọc nguồn, trung gian, đích lần lượt là cọc A, B, C, gọi n là số đĩa cần di chuyển. Đánh số đĩa từ 1(đĩa nhỏ nhất, trên cùng) đến n (đĩa lớn nhất, cuối cùng)

Để di chuyển n đĩa từ cột A sang cột C thì cần:

1. Chuyển (n-1) đĩa từ cọc A sang cọc B. Chỉ còn đĩa n ở cọc A.
2. Chuyển đĩa n từ cọc A sang cột C.
3. Chuyển đĩa (n-1) còn lại từ cọc B sang cọc C cho chúng nằm trên đĩa n.
4. Giải thuật không còn là đệ quy khi chỉ còn 1 đĩa, vì ta chỉ cần chuyển đĩa còn lại cuối cùng qua cọc đích mà không phải thông qua cọc trung gian.

Giải thuật được mô tả với số đĩa bằng 3 được minh họa như Hình 1.



Hình 1: Mô tả thuật toán đệ quy Tháp Hà Nội

1.1.3 Mã nguồn giải thuật đệ quy Tháp Hà Nội

Dưới đây là mã nguồn giải bài toán Tháp Hà Nội bằng mã lệnh Python

def thap_ha_noi(n,cot_a,cot_b,cot_c):
  if n==1:
  print(f'Chuyển đĩa {n} từ {cot_a} sang {cot_c} ')
  return
  thap_ha_noi(n-1,cot_a,cot_c,cot_b)
  print(f'Chuyển đĩa {n} từ { cot_a} sang {cot_c} ')
  thap_ha_noi(n-1,cot_b,cot_a,cot_c)
n=int(input('Nhap so dia:\t'))
thap_ha_noi(n,"a","b","c")
	- Tham số n là số đĩa cần di chuyển, cot_a, cot_b, cot_c lần lượt là cột nguồn, trung gian, đích.
	- Nếu mà chỉ có một đĩa thì chuyển từ cột nguồn sang cột đích.

Bảng 1: Mã nguồn bài toán Tháp Hà Nội

Dưới đây là kết quả bài toán tháp Hà Nội với số đĩa nhập vào tùy chọn như: Hình 2.

```
Chuyển đĩa 1 từ A sang C
Chuyển đĩa 2 từ A sang B
Chuyển đĩa 1 từ C sang B
Chuyển đĩa 3 từ A sang C
Chuyển đĩa 1 từ B sang A
Chuyển đĩa 2 từ B sang C
Chuyển đĩa 1 từ A sang C
```

Hình 2: Kết quả bài toán tháp Hà Nội với 3 đĩa

1.2 Bài toán tìm ước số chung lớn nhất

1.2.1 Giới thiệu về bài toán tìm ước số chung lớn nhất của hai số

Ước số chung lớn nhất của hai số a và b là k, điều kiện a và b đều chia hết cho k và k lớn nhất.

Để giải quyết bài toán này chúng ta có thể sử dụng nhiều cách làm. Em xin trình bày hai cách gồm: sử dụng đệ quy và sử dụng vòng lặp thông thường.

1.2.1 Giải quyết bài toán USCLN bằng đệ quy

def ucln(a, b):
  if a == 0 or b == 0:
  return a if b == 0 else b
  else:
  return ucln(b, a % b)
	- Đầu tiên kiểm tra hai số có bằng 0 hay khác 0, nếu cả hai số đều bằng 0 thì trả về None.
	- Nếu cả hai số đều khác 0 thì gọi hàm chính nó với tham số b và số dư của phép chia a chia b.

Bảng 2: Mã nguồn bài toán tìm USCLN bằng đệ quy

1.2.2 Giải quyết bài toán USCL bằng vòng lặp đơn giản

Để giải quyết bài toán tìm ước số chung lớn nhất của hai số, ngoài cách sử dụng đệ quy chúng ta còn có thể sử dụng vòng lặp đơn giản như dưới đây.

Trong bài toán này em sử dụng vòng lặp for để thực hiện.

def ucln(m,n):
  a=min(m,n)
  for i in rang(a,0,-1):
  if m%i==0 and n%i==0:
  return i
	- Sử dụng hàm có sẵn trong Python để tìm ra số bé hơn là a, vòng lặp for với biến i chạy từ số a về 1 với bước nhảy -1.
	- Khi m và n đều chia hết cho 1 số thì trả về kết quả và thoát khỏi vòng lặp.

Bảng 3: Mã nguồn bài toán tìm USCLN bằng vòng lặp

1.3 Bài toán mã đi tuần

1.3.1 Giới thiệu bài toán mã đi tuần

Mã đi tuần hay hành trình của quân mã là bài toán về việc chuyển một quân mã trên bàn cờ vua. Quân mã được đặt ở một ô trên một bàn cờ trống, nó phải di chuyển theo quy tắc của cờ vua để đi qua mỗi ô trên bàn cờ đúng một lần.

Nước đi của một quân mã giống hình chữ L và nó có thể di chuyển tất cả các hướng. Ở một vị trí thích hợp thì quân mã có thể di chuyển đến được 8 vị trí.

1.3.2 Mã nguồn bài toán mã đi tuần

Dưới đây là mã nguồn giải bài toán mã đi tuần như Bảng 4.

def kiem_tra_nuoc_di(x, y, board, n): return x>=0 and x<n and y>=0 and y<n and b oard[x][y]==-1	- Hàm kiem_tra_nuoc_di kiểm tra nước đi mới di chuyển có hợp lệ hay không, hợp lí thì trả về True và ngược lại.

def ma_di_chuyen(board, x, y, moves, movei, n) :
  if movei == n*n:
  return True
  for i in range(8):
  next_x = x+ moves[i][0]
  next_y = y+moves[i][1]
  if kiem_tra_nuoc_di(next_x, next_y, board, n)==True:
  board[next_x][next_y] = movei
  if ma_di_chuyen(board, next_x, next_y, moves, movei+1, n)==True:
  return True
  board[next_x][next_y] = -1
  return False
	- Tạo hàm ma_di_chuyen để tìm đường đi cho quân mã
	- Nếu số bước di chuyển của quân mã mà bằng số ô trên bàn cờ thì dừng lại.
	- Lặp qua tất cả các các bước có thể đi của quân mã, next_x và next_y lần lượt là tọa độ trên bàn cờ.
	- Sử dụng hàm kiem_tra_nuoc_di nếu đúng thì bước đi ở ô đó đánh số thứ tự của bước đi.
	- Thực hiện đệ quy hàm ma_di_chuyen như trước đó

def in_ket_qua(board):
  for i in range(len(board)):
  for j in range(len(board)):
	- Hàm in_ket_qua có nhiệm vụ in ra bài toán đã giải quyết.

print(board[i][j], end = ' ')
print()
 	
def kiem_tra_ket_qua(n):
board = [[1 for i in range(n)] for j in range(n)]
moves = [(2,1),(1,2),(-1,2),(-2,1),(-2,-1),(-1,-2),(1,-2),(2,-1)]
board[0][0] = 0
if ma_di_chuyen(board, 0, 0, moves, 1, n)==False:
print("Không tìm thấy giải pháp")
else:
in_ket_qua(board)
n=int(input('Nhập kích cỡ bàn cờ:'))
kiem_tra_ket_qua(n)
	- Hàm kiem_tra_ket_qua , tạo ma bàn cờ mô hình với các vị trí trên bàn cờ với giá trị bằng -1.
	- Gán vị trí bắt đầu với giá trị bằng 0
	- moves là list gồm các nước mà quân mã có thể di chuyển trên bàn cờ.
	- Nếu hàm ma_di_chuyen trả về False thì không có giải pháp cho bài toán và ngược lại thì gọi hàm in_ket_qua.

Bảng 4: Mã nguồn bài toán mã đi tuần

Dưới đây là kết quả của bài toán mã đi tuần với tùy chọn kích cỡ bàn cờ tùy ý như hình: Hình 3.

```
Nhập kích cỡ bàn cờ:8
0 59 38 33 30 17 8 63
37 34 31 60 9 62 29 16
58 1 36 39 32 27 18 7
35 48 41 26 61 10 15 28
42 57 2 49 40 23 6 19
47 50 45 54 25 20 11 14
56 43 52 3 22 13 24 5
51 46 55 44 53 4 21 12
```

Hình 3: Kết quả bài toán mã đi tuần với bàn cờ 8x8

1.4 Bài toán tám quân hậu

1.4.1 Giới thiệu bài toán tám quân hậu

Bài toán yêu cầu đặt tám quân hậu trên bàn cờ để làm sao không có quân hậu nào có thể bị tấn công. Các nước đi của quân hậu hoàn toàn như trong cờ vua.

1.4.2 Mã nguồn bài toán tám quân hậu

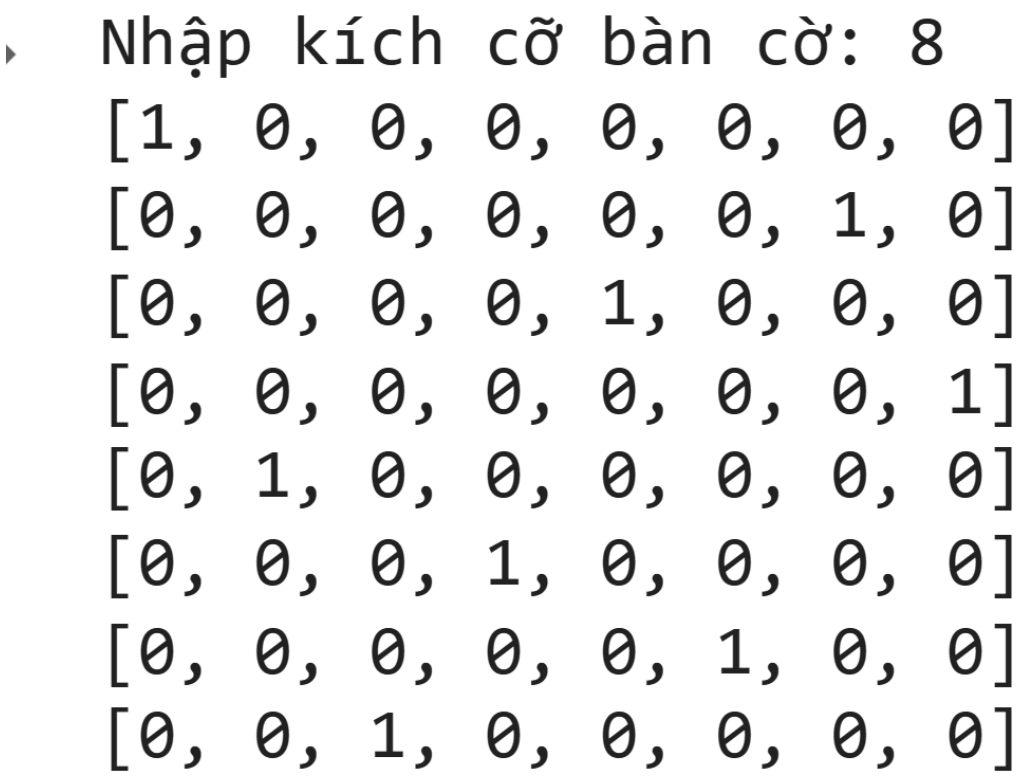
Dưới đây là mã nguồn giải bài toán tám quân hậu như Bảng 5.

def kiem_tra(board, row, col):
  for i in range(col):
  if board[row][i] == 1:
  return False
 	
  i, j = row, col
  while i >= 0 and j >= 0:
  if board[i][j] == 1:
  return False
  i -= 1
  j -= 1
 	
  i, j = row, col
  while i < len(board) and j >= 0:
  if board[i][j] == 1:
  return False
  i += 1
  j -= 1
 	
  return True
	- Kiểm tra hàng ngang, nếu trong hàng đã có đặt quân hậu rồi thì trả về False.
	
	- Mã lệnh phía bên sử dụng vòng lặp kiểm tra quân hậu đang tìm vị trí có bị tấn công theo chiều phía trên bên trái, nếu bị tấn công tiếp tục thử ô khác trong cột.
	
	- Mã lệnh phía bên sử dụng vòng lặp để kiểm tra quân hậu đang tìm vị trí có bị tấn công theo chiều phía dưới bên trái, nếu bị tấn công trả về false và quay lại tiếp tục thử các ô khác trong cột.
	- Nếu vị trí thỏa mãn thì đặt quân hậu vào vị trí đó

def tim_kiem(board, col):
  if col == len(board):
  return True
  for row in range(len(board)):
  if kiem_tra(board, row, col)==True:
  board[row][col] = 1
  if tim_kiem(board, col+1)==True:
  return True
  board[row][col] = 0
  return False
	- Hàm tim_kiem được sử dụng để tìm kiếm vị trí đặt các quân hậu
	- Nếu đã đặt được quân hậu vào cột cuối cùng thành công thì kết thúc bài toán.
	- Sử dụng vòng lặp để kiểm tra vị trí mới có thỏa mãn hay không.
	- Nếu vị trí mới thỏa mãn thì đệ quy đến cột tiếp theo.
	- Nếu không tìm thấy giải pháp thì quay lui tìm giải pháp mới.
	- Nếu không tìm được giải pháp thì trả về kết quả.

Bảng 5: Mã nguồn bài toán tám quân hậu

Dưới đây là kết quả bài toán 8 quân hậu trên bàn cờ, với số 1 là kí hiệu vị trí đặt quân hậu như: Hình 4.



Hình 4: Kết quả bài toán 8 quân hậu.

CHƯƠNG 2: DANH SÁCH LIÊN KẾT

2.1 Danh sách liên kết đơn

2.1.1 Giới thiệu danh sách liên kết đơn

Danh sách liên kết đơn(Single linked list): Chỉ có sự kết nối từ phần tử phía trước tới phần tử phía sau. Đây cũng là ví dụ tốt nhất và đơn giản nhất về cấu trúc dữ liệu động sử dụng con trỏ để cài đặt.

2.1.2 Cài đặt danh sách liên kết đơn

Dưới đây là mã nguồn cài đặt danh sách liên kết đơn như Bảng 6.

class Node:
  def __init__(self, data):
  self.data = data
  self.next = None
	- Lớp Node được tạo ra để đại diện cho một nút trong danh sách liên kết.
	- Gồm hai thuộc tính: data để lưu dữ liệu và next để lưu trữ tham chiếu đến nút kế tiếp trong danh sách liên kết.

class Dslk:
  def __init__(self):
  self.dau=None
  self.duoi=None
	- Lớp Dslk để tạo và quản lí một danh sách liên kết, gồm nhiều phương thức như : tìm kiếm, chèn, xóa,...
	- Phương thức khởi tạo gồm 2 thuộc tính: self.dau dùng để tham chiếu đến nút đầu tiên, self.duoi dùng để tham chiếu đến nút cuối cùng trong danh sách liên kết.

def add_LinkedList(self, data):
  node=Node(data)
  if self.dau is None:
  self.dau=node
  self.duoi=node
  else:
  self.duoi.next=node
  self.duoi=node
	- Phương thức add_LinkedList dùng để thêm 1 nút vào danh sách liên kế.
	- Nếu nút đầu là rỗng thì nút được thêm vào sẽ là nút đầu tiên.
	
	- Ngược lại ta thêm nút mới vào cuối danh sách bằng cách gán ‘next’ của nút cuối cùng danh sách hiện tại bằng nút mới, sau đó cập nhật nút cuối bằng nút mới thêm vào

def insert_LinkedList(self,index,value):
  node=Node(value)
  before=None
  now=self.dau
  i=0
  while i<index and now is not None:
  i+=1
  before=now
  now=now.next
  if before==None:
  node.next=self.dau
	- Phương thức chèn giá trị vào danh sách liên kết
	- Biến before gán bằng rỗng, biến now tham chiếu đến nút đầu tiên trong danh sách liên kết.
	- Sử dụng vòng lặp để tham chiếu đến nút vị thứ cần chèn, sau khi duyệt hết.
	- Nếu before rỗng có nghĩa là nút đầu tiên rỗng nên sẽ chèn vào đầu danh sách.

self.dau=node
if self.duoi==None:
self.duoi=node
else:
if now== None:
self.duoi.next=node
self.duoi=node
else:
before.next=node
node.next=now
	- Ngược lại, now đang trỏ đến vị trí kế tiếp mà rỗng thì có nghĩa chèn vào cuối danh sách. Ngược lại now không rỗng thì chèn vào giữa danh sách.

def find_LinkedList(self,data):
now=self.dau
index=0
a=[]
while now!= None:
if now.data==data:
a.append(index)
now=now.next
index +=1
return a
	- Phương thức tìm kiếm trong danh sách liên kết.
	- Biến now đang trỏ về nút đầu tiên của danh sách, index có ý nghĩa đánh số vị trí trong danh sách,
	- List a để chứa các vị trí có giá trị cần tìm
	- Lặp cho đến khi nút hiện tại là None, nếu nút data của nút hiện tại bằng giá trị cần tìm thì thêm index vào list a.
	- Cập nhật lại con trỏ trỏ đến nút tiếp theo, i đồng thời sau mỗi vòng lặp tăng lên 1.
	- Cuối cùng phương thức trả về danh sách các vị trí chứa giá trị cần tìm kiếm.

def remove_LinkedList(self,data):
if self.dau is None:
return
if self.dau.data == data:
self.dau = self.dau.next
now = self.dau
prev = None
while now is not None:
if now.data == data:
prev.next = now.next
prev = now
now = now.next
	- Phương thức xóa giá trị trong danh sách liên kết.
	- Nếu nút đầu tiên là None thì danh sách trống, không có giá trị để xóa.
	- Nếu nút đầu tiên bằng với giá trị cần xóa thì cập nhật lại nút đầu tiên mới là nút kế tiếp trong danh sách.
	- now đang trỏ tới nút đầu danh sách
	- Lặp cho đến nút cuối cùng nếu giá trị của nút hiện tại bằng với giá trị cần xóa thì cập nhật trỏ tiếp theo của nút trước đó trỏ tới nút kế tiếp của nút hiện tại, có nghĩa là bỏ qua nút hiện tại.
	- Tiếp tục cập nhật prev trỏ nút hiện tại, còn now trỏ đến nút tiếp theo trong danh sách.

def print_LinkedList(self): print('Danh sách liên kết:',end=' ') temp=self.dau	- Phương thức in danh sách liên kết. - temp biến tạm trỏ đến nút đầu tiên của danh sách liên kết.

while (temp): print(temp.data,end=' ') temp=temp.next	- Lặp cho đến khi nút hiện tại là trống, đồng thời in giá trị của nút hiện tại cách nhau bằng khoảng trắng. - Cập nhật con trỏ hiện tại trỏ tới nút tiếp theo sau mỗi vòng lặp.

def main():
ds=Dslk()
ds.add_LinkedList(int(input('Thêm giá trị vào danh sách:')))
ds.add_LinkedList(int(input('Thêm giá trị vào danh sách:')))
ds.add_LinkedList(int(input('Thêm giá trị vào danh sách:')))
ds.add_LinkedList(int(input('Thêm giá trị vào danh sách:')))
ds.add_LinkedList(int(input('Thêm giá trị vào danh sách:')))
index,value = map(int, input('Nhập lần lượt vị trí và giá trị muốn chèn (phân tách bằng khoảng trắng): ').split(" "))
ds.insert_LinkedList(index,value)
ds.add_LinkedList(int(input('Thêm giá trị vào danh sách:')))
print(ds.find_LinkedList(int(input('Nhập giá trị muốn tìm kiếm trong danh sách:'))))
ds.insert_LinkedList(7,18)
ds.remove_LinkedList(int(input('Nhập giá trị muốn xóa khỏi danh sách liên kết:')))
ds.print_LinkedList()
if __name__ =='__main__':
main()
	- Hàm chính thực hiện các thao tác với danh sách liên kết.

Bảng 6: Mã nguồn danh sách liên kết đơn

Sau khi thực thi hàm chính với danh sách liên kết kết quả như: Hình 5.

```
Thêm giá trị vào danh sách:0
Thêm giá trị vào danh sách:2
Thêm giá trị vào danh sách:3
Thêm giá trị vào danh sách:5
Thêm giá trị vào danh sách:8
Nhập lần lượt vị trí và giá trị muốn chèn (phân tách bằng khoảng trắng): 1 1
Thêm giá trị vào danh sách:7
Nhập giá trị muốn tìm kiếm trong danh sách:5
[4]
Nhập giá trị muốn xóa khỏi danh sách liên kết:3
Danh sách liên kết: 0 1 2 5 8 7 18
```

Hình 5: Kết quả thực thi với danh sách liên kết đơn

2.2 Danh sách liên kết đôi

2.2.1 Giới thiệu danh sách liên kết đôi

Danh sách liên kết đôi (hay còn gọi là danh sách liên kết kép) là một cấu trúc dữ liệu được sử dụng trong lập trình để lưu trữ và quản lý một danh sách các phần tử, trong đó mỗi phần tử bao gồm hai phần: một giá trị và hai con trỏ, mỗi con trỏ trỏ tới phần tử phía trước và phía sau của phần tử đó trong danh sách.

2.2.2 Cài đặt danh sách liên kết đôi

Dưới dây là mã nguồn cài đặt danh sách liên kết đôi như: Bảng 7.

class Node:
  def __init__(self, data):
  self.data = data
  self.prev = None
  self.next = None
	- Tạo lớp nút
	- Phương thức khởi tạo, self.data nhận làm giá trị của một nút, self.prev con trỏ trỏ tới nút trước đó còn self.next trỏ tới nút kế tiếp.

class DoublyLinkedList:
  def __init__(self):
  self.head = None
	- Tạo lớp danh sách liên kết đôi
	- self.head con trỏ nút đầu tiên của danh sách

def append(self, data):
  new_node = Node(data)
  if self.head is None:
  self.head = new_node
  else:
  current = self.head
  while current.next!=None:
  current = current.next
  current.next = new_node
  new_node.prev=current
	- Phương thức thêm nút mới vào danh sách liên kết
	- Biến new_node sẽ nhận giá trị thêm vào làm nút mới.
	- Nếu nút đầu tiên trống thì nhận nút vừa thêm vào làm nút đầu tiên.
	- Ngược lại biến current trỏ tới nút đầu tiên, lặp liên tục đến cuối danh sách. Khi đó current đang trỏ về nút cuối cùng, nút tiếp theo sẽ được thêm vào
	- Nút mới trỏ về nút hiện tại làm nút phía trước.

insert(self,data,index): new_node=Node(data)	

now=self.head
temp=None
i=0
while i<index and now!= None:
i+=1
temp=now
now=now.next
 	
if temp==None:
new_node.next=self.head
self.head=new_node
else:
temp.next=new_node
new_node.next=now
 	
def delete(self, data):
temp=self.head
while temp is not None:
if temp.data==data:
if temp.prev is None:
self.head = temp.next
 	
else:
if temp.next is not None:
temp.prev.next=temp.next
temp.next.prev=temp.prev
else:
temp.prev.next=None
temp=temp.next
 	
def search(self,data):
i=0
a=[]
temp=self.head
while temp is not None:
if temp.data==data:
a.append(i)
i+=1
	- Phương thức chèn nút mới vào danh sách liên kết đôi.
	- Biến new_node nhận giá trị thêm vào làm nút mới
	
	- Lặp đến vị trí cần chèn hoặc nút cuối cùng.
	
	- Kết thúc vòng lặp nếu temp trống thì danh sách liên kết này rỗng vì thế sẽ nhận làm nút đầu tiên
	- Ngược lại nếu vị trí muốn thêm vào vượt quá danh sách thì thêm vào cuối danh sách liên kết, còn không thì thêm vào giữa danh sách.
	
	- Phương thức xóa nút trong danh sách liên kết đôi
	- Biến temp ban đầu trỏ đến nút đầu tiên của danh sách.
	- Lặp đến nút cuối cùng, nếu nút hiện tại bằng với giá trị muốn xóa:
	+ Nếu nút trươc đó trống có nghĩa hiện tại là nút đầu tiên vì thế nút đầu tiên sẽ cập nhật lại là nút tiếp theo.
	- Ngược lại, nếu nút trước không trống, nút bị xóa không phải là nút cuối thì con trỏ tiếp theo của nút trước sẽ trỏ đến nút kế tiếp, con trỏ trước của nút tiếp theo sẽ trỏ đến nút trước đó.
	- Nếu nút bị xóa là nút cuối thì con trỏ của nút tiếp theo của nút trước đó sẽ trỏ về None
	
	- Phương thức tìm kiếm với giá trị đầu vào
	- Biến i đánh dấu vị trí giá trị
	- List a dùng để chứa các vị trí của giá trị cần tìm.

temp=temp.next
if a is None: return None
else: return a
 	
def display(self):
current = self.head
while current:
print(current.data,end=' ')
current = current.next
	- Lặp đến nút cuối cùng, nếu giá trị của nút nào bằng với giá trị cần tìm thì thêm vị trí đó vào list a
	- Sau mỗi vòng lặp thì giá trị i tăng lên 1 và nút hiện tại sẽ trỏ tới nút kế tiếp
	- Kết thúc quá trình tìm kiếm nếu list a trống thì trả về None, ngược lại thì trả về danh sách vị trí.
	
	- Phương thức hiển thị danh sách liên kết, current trỏ đến nút đầu tiên.
	- Lặp cho đến khi hết danh sách, trong quá trình lặp thì in ra giá trị các nút cách nhau bởi khoảng trắng, sau khi in ra nút hiện tại thì tiếp tục trỏ đến nút tiếp theo

def main():
a=1
ds=DoublyLinkedList()
while a==1:
data=int(input('Nhập giá trị bạn muốn thêm vào danh sách: '))
ds.append(data)
a=int(input('Nhập số 1 để thêm giá trị mới vào danh sách: '))
data=int(input('Nhập giá trị bạn muốn chèn: '))
index=int(input('Nhập vị trí bạn muốn chèn: '))
ds.insert(data,index)
print(ds.search(int(input('Nhập giá trị bạn muốn tìm kiếm: '))))
ds.delete(int(input('Nhập giá trị bạn muốn xóa khỏi danh sách liên kết: ')))
print('Danh sách liên kết:')
ds.display()
if __name__=="__main__":
main()
	- Tạo chương trình chính
	- Thực hiện thao tác với các phương thức của lớp danh sách liên kết đôi.

Bảng 7: Mã nguồn cài đặt danh sách liên kết đôi

```
Nhập giá trị bạn muốn thêm vào danh sách: 0
Nhập số 1 để thêm giá trị mới vào danh sách: 1
Nhập giá trị bạn muốn thêm vào danh sách: 1
Nhập số 1 để thêm giá trị mới vào danh sách: 1
Nhập giá trị bạn muốn thêm vào danh sách: 2
Nhập số 1 để thêm giá trị mới vào danh sách: 1
Nhập giá trị bạn muốn thêm vào danh sách: 3
Nhập số 1 để thêm giá trị mới vào danh sách: 1
Nhập giá trị bạn muốn thêm vào danh sách: 4
Nhập số 1 để thêm giá trị mới vào danh sách: 0
Nhập giá trị bạn muốn chèn: 5
Nhập vị trí bạn muốn chèn: 2
Nhập giá trị bạn muốn tìm kiếm: 3
[4]
Nhập giá trị bạn muốn xóa khỏi danh sách liên kết: 3
Danh sách liên kết:
0 1 5 2 4
```

Hình 6: Kết quả thực thi với danh sách liên kết đôi

2.3 Ngăn xếp (Stack)

2.3.1 Giới thiệu ngăn xếp (Stack)

Ngăn xếp (stack) là một cấu trúc dữ liệu trừu tượng, được sử dụng để lưu trữ một tập hợp các phần tử. Các phần tử được lưu trữ theo cách xếp chồng lên nhau, trong đó phần tử cuối cùng được thêm vào (còn gọi là đỉnh của stack) là phần tử đầu tiên được lấy ra.

2.3.2 Cài đặt ngăn xếp (Stack)

Dưới đây là mã nguồn cài đặt ngăn xếp – (Stack) như Bảng 8.

class Stack:
  def __init__(self):
  self.items=[]
 	
  def push(self,item):
  self.items.append(item)
 	
  def __len__(self):
  return (f'Độ dài ngăn sắp xếp: {len(self.items)}')
 	
  def pop(self):
	- Tạo lớp Stack
	- Phương thức khởi tạo danh sách liên kết rỗng.
	
	- Phương thức push dùng để thêm phần tử vào danh sách liên kết.
	
	- Mã lệnh phía bên trả về độ dài của ngăn xếp.

`if len(self.items==0): return " stack is empty"`
`return self.items.pop()`
 	
`def stack_is_empty(self):`
`if len(self.items)==0:`
`return "Ngăn xếp rỗng"`
`else: return "stack is not empty "`
 	
`def __str__(self):`
`return str(self.items)`
 	
`ds=Stack()`
`ds.push(int(input('Thêm phần tử vào ngăn xếp: ')))`
`ds.push(int(input('Thêm phần tử vào ngăn xếp: ')))`
`ds.push(int(input('Thêm phần tử vào ngăn xếp: ')))`
`ds.push(int(input('Thêm phần tử vào ngăn xếp: ')))`
`print('Phần tử được lấy ra đầu tiên: ',ds.pop())`
`ds.stack_is_empty()`
`print('Độ dài ngăn xếp',ds.__len__())`
`print('Ngăn xếp:',ds)`
	- Phương thức pop lấy ra phần tử cuối cùng của ngăn xếp, nếu ngăn xếp rỗng thì thông báo ngăn xếp rỗng.
	
	- Phương thức stack_is_empty kiểm tra ngăn xếp có phải là rỗng hay không.
	
	- Mã lệnh phía bên dùng để xuất ra màn hình các phần tử trong ngăn xếp.
	
	- Thao tác với ngăn xếp

Bảng 8: Mã nguồn cài đặt ngăn xếp-(Stack)

Dưới đây là kết quả sau khi thực hiện thao tác với ngăn xếp như: Hình 7.

```
Thêm phần tử vào ngăn xếp: 5
Thêm phần tử vào ngăn xếp: 3
Thêm phần tử vào ngăn xếp: 0
Thêm phần tử vào ngăn xếp: -6
Phần tử được lấy ra đầu tiên:  -6
Độ dài ngăn xếp Độ dài ngăn sắp xếp: 3
Ngăn xếp: [5, 3, 0]
```

Hình 7: Kết quả thao tác với ngăn xếp

2.4 Hàng đợi (Queue)

2.4.1 Giới thiệu hàng đợi (Queue)

Hàng đợi (Queue) là một cấu trúc dữ liệu dùng để chứa các đối tượng theo cơ chế FIFO (First In First Out) có nghĩa là vào trước ra sau.

Trong hàng đợi, các đối tượng được thêm vào bất cứ lúc nào nhưng khi lấy ra chỉ được phép lấy ra phần tử đầu tiên trong hàng đợi. Việc thêm đối tượng vào hàng đợi luôn diễn ra ở cuối hàng đợi, ngược lại việc lấy đối tượng ra khỏi hàng đợi luôn diễn ra ở đầu hàng đợi.

2.4.2 Cài đặt hàng đợi (Queue)

Dưới đây là mã nguồn cài đặt hàng đợi (Queue) như

class Queue:
  def __init__(self):
  self.items = []
	- Tạo lớp Queue
	- Phương thức khởi tạo hàng đợi rỗng.

def is_empty(self): return len(self.items) == 0	- Mã lệnh phía bên là phương thức kiểm tra hàng đợi có rỗng hay không, trả về đúng hoặc sai.
def push(self, item): self.items.append(item)	- Phương thức enqueue thêm phần tử vào cuối hàng đợi.

def pop(self):
  if self.is_empty():
  return None
  return self.items.pop(0)
	- Phương thức dequeue có vai trò lấy phần tử đầu tiên trong hàng đợi theo quy tắc FIFO (vào trước ra sau), nếu hàng đợi rỗng thì trả về None.

def __len__(self): return len(self.items)	- Phương thức len dùng để truy xuất độ dài hàng đợi hay còn gọi là số phần tử trong hàng đợi.

Bảng 9: Mã nguồn cài đặt hàng đợi(Queue)

Dưới đây là kết quả sau khi thao tác với hàng đợi như hình: Hình 8.

```
Thêm phần tử vào hàng đợi: 4
Thêm phần tử vào hàng đợi: 5
Thêm phần tử vào hàng đợi: 6
Thêm phần tử vào hàng đợi: 1
Phần tử được lấy ra đầu tiên:  4
Độ dài ngăn xếp 3
Ngăn xếp: [5, 6, 1]
```

Hình 8: Kết quả thao tác với hàng đợi

CHƯƠNG 3: CÂY

3.1 Giới thiệu cấu trúc dữ liệu cây

Cây lập trình (Programming Tree) là một cấu trúc dữ liệu mô tả cách mà các khối mã (code blocks) được tổ chức trong một chương trình hoặc hàm. Các khối mã được đặt trong các nút của cây và quan hệ giữa chúng được mô tả bởi các cạnh của cây. Cây lập trình thường được sử dụng để biểu diễn cấu trúc chương trình hoặc hàm một cách rõ ràng, giúp cho việc đọc và hiểu mã nguồn trở nên dễ dàng hơn.

Các loại cây lập trình phổ biến bao gồm cây cú pháp (Syntax Tree), cây thực thi (Execution Tree) và cây AST (Abstract Syntax Tree). Cây cú pháp biểu diễn cấu trúc cú pháp của chương trình, cây thực thi mô tả các bước thực thi của chương trình và cây AST biểu diễn cấu trúc trừu tượng của chương trình một cách rõ ràng và đơn giản hơn.

3.1.1 Ý nghĩa cấu trúc dữ liệu cây

Việc sử dụng cây lập trình mang lại nhiều lợi ích trong việc lập trình và phát triển phần mềm. Dưới đây là một số ý nghĩa của việc sử dụng cây lập trình:

- Biểu diễn cấu trúc chương trình một cách rõ ràng: Cây lập trình giúp biểu diễn cấu trúc của chương trình một cách rõ ràng và dễ hiểu hơn. Các khối mã được tổ chức theo một thứ tự nhất định, giúp cho việc đọc và hiểu mã nguồn trở nên dễ dàng hơn.
- Dễ dàng tìm lỗi: Cây lập trình cũng giúp cho việc tìm lỗi trong chương trình trở nên dễ dàng hơn. Với cây lập trình, các lỗi có thể được xác định và giải quyết nhanh chóng, giúp tiết kiệm thời gian và công sức trong việc sửa lỗi.
- Phát triển phần mềm dễ dàng hơn: Sử dụng cây lập trình giúp cho việc phát triển phần mềm trở nên dễ dàng hơn. Việc biểu diễn cấu trúc của chương trình một cách rõ ràng giúp cho các nhà phát triển dễ dàng hơn trong việc tạo, thay đổi và cải tiến chương trình.
- Dễ dàng đọc và hiểu mã nguồn: Cây lập trình giúp cho mã nguồn trở nên dễ đọc và hiểu hơn. Với cấu trúc rõ ràng, các nhà phát triển có thể dễ dàng hình dung được cấu trúc của chương trình mà không cần phải đọc toàn bộ mã nguồn.
- Tăng tính cấu trúc và tái sử dụng của chương trình: Sử dụng cây lập trình giúp tăng tính cấu trúc và tái sử dụng của chương trình. Các khối mã được tổ chức một cách rõ ràng, giúp cho việc sử dụng lại mã nguồn trở nên dễ dàng hơn, từ đó giúp tiết kiệm thời gian và công sức trong quá trình phát triển phần mềm.

3.1.2 Cách thức hoạt động cấu trúc dữ liệu cây

Cây lập trình hoạt động bằng cách sử dụng các nút và liên kết giữa chúng để biểu diễn cấu trúc của một chương trình máy tính. Mỗi nút trong cây lập trình đại diện cho một thực thể trong chương trình, ví dụ như một toán hạng, một biến, một phép tính, hoặc một lệnh điều khiển. Các liên kết giữa các nút thể hiện mối quan hệ logic giữa chúng.

Cây lập trình thường được xây dựng dựa trên ngôn ngữ lập trình và có thể được tạo ra bằng tay hoặc bằng các công cụ tự động. Cây lập trình được sử dụng trong nhiều môi trường lập trình khác nhau, bao gồm các trình biên dịch, trình thông dịch, trình gỡ lỗi, và các công cụ phân tích chương trình.

Khi một chương trình được thực thi, cây lập trình được duyệt theo từ trên xuống dưới bằng cách sử dụng các thuật toán duyệt cây như duyệt theo chiều sâu (depth-first)

hoặc duyệt theo chiều rộng (breadth-first). Khi duyệt cây, các nút sẽ được thực thi tuần tự theo thứ tự được quy định bởi cấu trúc của cây lập trình.

Mã nguồn cài đặt cây trong ngôn ngữ Python như: Bảng 10.

class Node:
  def __init__(self,key):
  self.key=key
  self.left=None
  self.right=None
  def insert_key(self,key):
  new_node=Node(key)
  if self.key is None:
  self.key=new_node.key
  else:
  if key < self.key:
  if self.left is None:
  self.left=new_node
  else: self.left.insert_key(key)
  elif key> self.key:
  if self.right is None:
  self.right=new_node
  else: self.right.insert_key(key)
  else:
  print(f'Khóa bị trùng: {key}')
	- Tạo lớp Node
	- Phương thức khởi tạo nút gốc có thể nhận giá trị hoặc trống, hai nút bên trái và phải trống.
	
	- Phương thức chèn với giá trị tùy chọn
	- Nếu nút gốc là trống thì nút mới được thêm vào đầu tiên sẽ nhận làm gốc.
	
	- Ngược lại, nếu nút gốc không trống thì giá trị thêm vào bé hơn nút gốc thì được thêm vào phía bên trái
	- Nếu giá trị được thêm vào lớn hơn gốc thì được thêm vào bên phải nút gốc
	- Nếu giá trị thêm vào mà trùng với nút gốc thì in ra lệnh thông báo trùng.

Bảng 10: Cài đặt cây

3.2 Duyệt cây theo thứ tự trước

3.2.1 Giới thiệu duyệt cây theo thứ tự trước

Duyệt cây theo thứ tự trước (pre-order traversal) là một cách duyệt qua tất cả các nút trong cây theo thứ tự nhất định. Khi duyệt cây theo thứ tự trước, trước tiên ta sẽ duyệt qua nút gốc, sau đó duyệt qua tất cả các nút con bên trái của nút gốc, và cuối cùng là tất cả các nút con bên phải của nút gốc.

3.2.2 Mã nguồn duyệt cây theo thứ tự trước

Dưới đây là mã nguồn duyệt cây theo thứ tự trước bằng mã lệnh Python như: Bảng 11.

def Preorder_Traversal(node):
  if node is None: return
  print(node.key,end=' ')
  Preorder_Traversal(node.left)
  Preorder_Traversal(node.right)
	- Tạo hàm duyệt cây theo thứ tự trước.
	- Nếu cây trống thì hàm trả về None.
	- Ngược lại, thì sẽ in ra nút gốc, tiếp theo đệ quy hàm duyệt cây theo thứ tự trước với tham số nút trái và nút phải.

Bảng 11: Mã nguồn duyệt cây theo thứ tự trước

3.3 Duyệt cây theo thứ tự sau

3.3.1 Giới thiệu duyệt cây theo thứ tự sau

Duyệt cây theo thứ tự sau (postorder traversal) là một phương pháp duyệt cây nhị phân để truy cập vào các nút theo thứ tự sau cùng. Trong phương pháp này, cây con trái được duyệt trước, sau đó là cây con phải và cuối cùng là nút gốc.

3.3.2 Mã nguồn duyệt cây theo thứ tự sau

Dưới đây là mã nguồn duyệt cây theo thứ tự sau bằng mã lệnh Python như: Bảng 12.

def Postoder_Traversal(node):
  if node is None: return
  Postoder_Traversal(node.left)
  Postoder_Traversal(node.right)
  print(node.key,end=' ')
	- Tạo hàm duyệt cây theo thứ tự sau.
	- Nếu cây trống trả về None.
	- Đệ quy hàm duyệt cây với tham số nút trái
	- Tiếp tục đệ quy với tham số nút phải
	- Sau khi đệ quy xong thì in nút gốc.

Bảng 12: Mã nguồn duyệt cây theo thứ tự sau

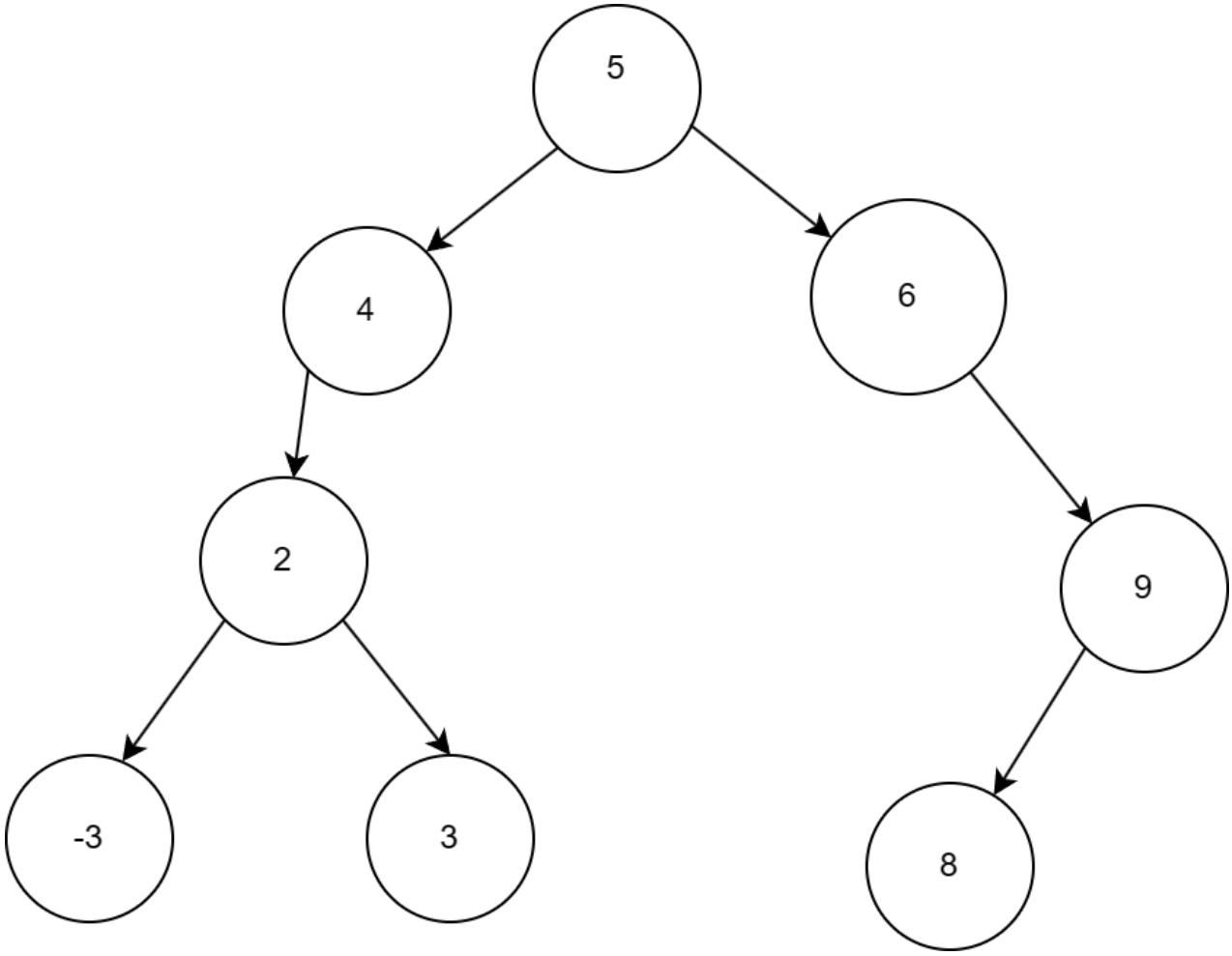
3.4 Chương trình chính thực hiện duyệt cây

Chương trình chính thực hiện duyệt cây theo thứ tụ trước và sau như: Bảng 13.

def main():
  tree=Node(None)
  a=int(input('Nhập số 1 để thêm nút vào cây:'))
  while a==1:
  tree.insert_key(int(input('Nhập nút muốn thêm:')))
  a=int(input('Nhập số 1 để thêm nút vào cây:'))
  print('Duyệt cây theo thứ tự trước:')
  Preorder_Traversal(tree)
  print('Duyệt cây theo thứ tự sau:')
  Postoder_Traversal(tree)
if __name__ == '__main__':
  main()
	- Tạo hàm chương trình chính.
	- Khởi tạo ban đầu cây rỗng.
	- Dùng vòng lặp để thuận tiện cho việc thêm nút vào cây.
	-Sau khi kết thúc quá trình thêm nút vào cây thì in kết quả.

Bảng 13: Mã nguồn chương trình chính thực hiện duyệt cây

Dưới đây là hình ảnh mô tả cây khi thực hiện chương trình chính như: Hình 9.



Hình 9: Mô tả cây

Kết quả sau khi thực hiện chương trình chính như :Hình 10.

```
Nhập số 1 để thêm nút vào cây:1
Nhập nút muốn thêm:5
Nhập số 1 để thêm nút vào cây:1
Nhập nút muốn thêm:4
Nhập số 1 để thêm nút vào cây:1
Nhập nút muốn thêm:6
Nhập số 1 để thêm nút vào cây:1
Nhập nút muốn thêm:2
Nhập số 1 để thêm nút vào cây:1
Nhập nút muốn thêm:-3
Nhập số 1 để thêm nút vào cây:1
Nhập nút muốn thêm:3
Nhập số 1 để thêm nút vào cây:1
Nhập nút muốn thêm:9
Nhập số 1 để thêm nút vào cây:1
Nhập nút muốn thêm:8
Nhập số 1 để thêm nút vào cây:0
Duyệt cây theo thứ tự trước:
5 4 2 -3 3 6 9 8
Duyệt cây theo thứ tự sau:
-3 3 2 4 8 9 6 5
```

Hình 10: Kết quả duyệt cây theo thứ tự trước và sau

CHƯƠNG 4: ĐỒ THỊ

4.1 Đồ thị vô hướng

4.1.1 Nội dung

Đồ thị vô hướng là một loại đồ thị trong đó các cạnh không có hướng đi riêng biệt, tức là các cạnh không có phân biệt giữa đỉnh đầu và đỉnh cuối. Ví dụ, nếu có một cạnh kết nối hai đỉnh A và B trong đồ thị vô hướng, thì ta có thể đi từ A đến B hoặc từ B đến A bằng cùng một cạnh. Các đỉnh của đồ thị vô hướng cũng không có hướng đi riêng biệt, có nghĩa là nếu hai đỉnh nối với nhau bằng một cạnh thì chúng được coi là kề nhau.

4.1.2 Ý nghĩa

Đồ thị vô hướng thường được sử dụng để mô hình hóa mối quan hệ giữa các đối tượng hoặc sự kiện trong thực tế, ví dụ như mạng xã hội, đường phố, hệ thống giao thông và các mối liên kết trong phân tích mạng lưới

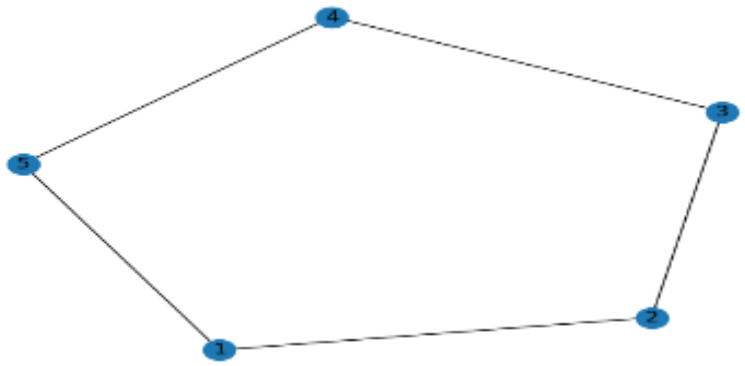
4.1.3 Mã nguồn cài đặt đồ thị vô hướng

Dưới đây là mã nguồn cài đặt đồ thị có hướng như: Bảng 14.

import networkx as nx import matplotlib.pylot as plt G= nx.Graph() G.add_nodes_from([1,2,3,4,5]) G.add_edges_from([(1,2),(2,3),(3,4),(3,4),(4,5)]) nx.draw(G, with_lables= True) plt.show()	- Cài đặt, khai báo các thư viện cần thiết. - Tạo đồ thị - Thêm các đỉnh vào đồ thị - Thêm các cạnh vào đồ thị - Vẽ đồ thị - Hiển thị đồ thị ra màn hình

Bảng 14: Mã nguồn cài đặt đồ thị vô hướng

Dưới đây là kết quả bài toán đồ thị vô hướng như: Hình 11.



Hình 11: Kết quả bài toán đồ thị vô hướng

4.2 Đồ thị có hướng

4.2.1 Nội dung

Đồ thị có hướng (directed graph) là một loại đồ thị mà các cạnh có hướng đi từ một đỉnh (đỉnh gốc) đến đỉnh khác (đỉnh đích). Các cạnh trong đồ thị có hướng được gọi là cạnh đi và cạnh đến, và thường được biểu diễn bằng mũi tên.

4.2.2 Ý nghĩa

Đồ thị có hướng (directed graph) được sử dụng để mô hình hóa các quan hệ có hướng, tức là các quan hệ mà có sự tương tác giữa các đối tượng theo một hướng nhất định. Ví dụ như mô hình hệ thống định tuyến mạng, các mô hình quan hệ và liên kết giữa các trang web, hoặc các mô hình thực hiện truyền tin trên mạng xã hội.

Trong các ứng dụng thực tế, đồ thị có hướng được sử dụng để giải quyết các bài toán phức tạp như định tuyến mạng, phân tích quan hệ giữa các đối tượng, tìm kiếm đường đi ngắn nhất và phân tích mạng xã hội. Ngoài ra, đồ thị có hướng cũng được sử dụng trong các lĩnh vực như lý thuyết điều khiển, xử lý ngôn ngữ tự nhiên và sinh học tính toán.

4.2.3 Mã nguồn cài đặt đồ thị có hướng

Dưới đây là mã nguồn cài đặt đồ thị có hướng như: Bảng 15.

import matplotlib.pyplot as plt import networkx as nx import pprint	- Cài đặt các thư viện cần thiết

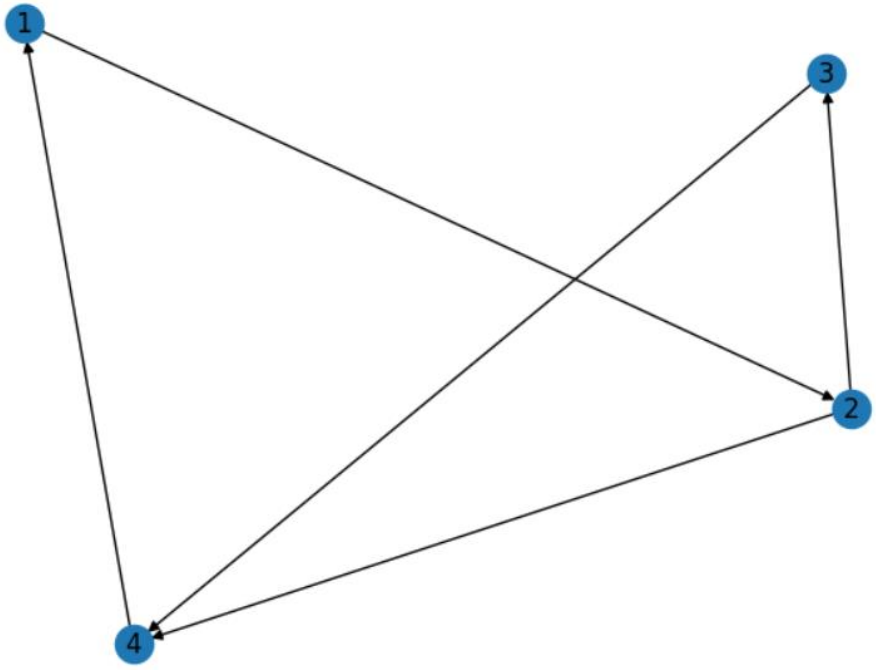
def nhap_ma_tran_ke():
  N = int(input("Nhập số đỉnh của đồ thị: "))
  adj = [[0] * (N + 1) for _ in range(N + 1)]
  M = int(input("Nhập số cạnh của đồ thị: "))
  print("Nhập các cạnh của đồ thị (u, v):")
  for i in range(M):
  u, v = map(int, input().split())
  adj[u][v] += 1
  return adj
	- Hàm khởi tạo ma trận
	- Khởi tạo ma trận kề kích thước (N+1) x (N+1)
	- Tăng số cạnh từ u đến v (Hàng u cột v) (Giá trị 0 -> 1)

def ve_do_thi(ma_tran_ke):
  G = nx.DiGraph()
  N = len(ma_tran_ke) - 1
 	
  for u in range(1, N + 1):
  for v in range(1, N + 1):
  if ma_tran_ke[u][v] > 0:
  G.add_edge(u, v)
 	
  pos = nx.spring_layout(G)
  nx.draw(G, pos, with_labels=True, arrows=True)
  plt.show()
	- Tạo đồ thị có hướng
	- Thêm cạnh từ u đến v
	- Vẽ đồ thị

Bảng 15: Mã nguồn cài đặt đồ thị có hướng

Dưới đây là kết quả bài toán đồ thị có hương như: Hình 12.

```
Nhập số đỉnh của đồ thị: 4
Nhập số cạnh của đồ thị: 5
Nhập các cạnh của đồ thị (u, v):
1 2
2 3
3 4
4 1
2 4
[[0, 0, 0, 0, 0],
 [0, 0, 1, 0, 0],
 [0, 0, 0, 1, 1],
 [0, 0, 0, 0, 1],
 [0, 1, 0, 0, 0]]
```



Hình 12: Kết quả bài toán đồ thị có hướng

CHƯƠNG 5: SẮP XẾP

5.1 Sắp xếp chọn

5.1.1 Giới thiệu về sắp xếp chọn

Sắp xếp chọn là một thuật toán sắp xếp đơn giản, dựa trên việc so sánh tại chỗ. trong đó danh sách được chia thành hai phần, phần được sắp xếp (sorted list**)** ở bên trái và phần chưa được sắp xếp **(**unsorted list) ở bên phải. Ban đầu, phần được sắp xếp là trống và phần chưa được sắp xếp là toàn bộ danh sách ban đầu.

5.1.2 Cài đặt sắp xếp chọn

Dưới đây là mã nguồn sắp xếp chọn (tăng dần) như bảng Bảng 8.

def sap_xep_chon(lst):
  lst_1=lst.copy()
  for i in range(len(lst)-1):
  min=i
  for j in range(i+1,len(lst_1)):
  if lst_1[j]<lst_1[min]:
  min=j
  lst_1[min],lst_1[i]=lst_1[i],lst_1[min]
  return lst_1
 	
n=int(input('Nhập số phần tử của mảng: '))
lst=[]
for i in range (0,n,1):
  lst.append(int(input('Nhập giá trị: ')))
print('Mảng chưa sắp xếp: ', lst)
print('Mảng đã sắp xếp:',sap_xep_chon(lst))
	- Dựng hàm sap_xep_chon với tham số truyền vào là 1 list số thực.
	- Tạo một bản sao của list đó để có thể tái sử dụng.
	- Sử dụng vòng lặp lồng vòng lặp, gán biến min=i ở vị trí đầu tiền,sau đó sử dụng vòng lặp bắt đầu từ vị trí thứ i+1 đến phần tử cuối cùng, nếu gặp phần tử bé hơn vị trí min thì gán lại min bằng vị trí đó.
	- Sau khi ra khỏi vòng lặp thì hoán đổi vị trí min ban đầu với vị trí min vừa tìm thấy (lúc này trị trí có giá trị min sẽ lên đầu danh sách)

Bảng 16: Mã nguồn sắp xếp chọn

Dưới đây là kết quả sau khi thực hiện chương trình sắp xếp chọn như

```
Nhập số phần tử của mảng: 5
Nhập giá trị: 2
Nhập giá trị: 0
Nhập giá trị: 1
Nhập giá trị: -3
Nhập giá trị: 4
Mảng chưa sắp xếp:  [2, 0, 1, -3, 4]
Mảng đã sắp xếp: [-3, 0, 1, 2, 4]
```

Hình 13: Kết quả sắp xếp chọn

5.2 Sắp xếp chèn

5.2.1 Giới thiệu sắp xếp chèn

Sắp xếp chèn là một giải thuật sắp xếp dựa trên so sánh in-place. Ở đây, một danh sách con luôn luôn được duy trì dưới dạng đã qua sắp xếp. Sắp xếp chèn là chèn thêm một phần tử vào danh sách con đã qua sắp xếp. Phần tử được chèn vào vị trí thích hợp sao cho vẫn đảm bảo rằng danh sách con đó vẫn sắp theo thứ tự.

5.2.2 Cài đặt sắp xếp chèn

Dưới đây là mã nguồn sắp xếp chèn (tăng dần) như Bảng 17.

def sap_xep_chen(lst):
  lst_1=lst.copy()
  for i in range(len(lst)):
  index_min=i
  min=lst_1[i]
  while(index_min>0 and(lst_1[index_min-1]>min)):
  lst_1[index_min]=lst_1[index_min-1]
  index_min-=1
  lst_1[index_min]=min
 	
  return lst_1
n=int(input('Nhập số phần tử của mảng: '))
lst=[]
for i in range (0,n,1):
  lst.append(int(input('Nhập giá trị: ')))
print('Mảng chưa sắp xếp: ', lst)
print('Mảng đã sắp xếp:',sap_xep_chen(lst))
	- Dựng hàm sap_xep_chen với tham số lst là 1 list số thực.
	- Gán vị trí có giá trị nhỏ nhất là phần tử đầu tiên.
	- Nếu index_min >0 và giá trị nằm trước phần tử nhỏ nhất lớn hơn min thì gán lại phần tử có giá trị nhỏ nhất bằng phần tử trước đó, sau khi thoát khỏi vòng lặp giá trị sẽ được cập nhật lại. Nếu không qua vòng lặp thì sẽ vẫn giữ nguyên.

Bảng 17: Mã nguồn sắp xếp chèn

Dưới đây là kết quả sắp xếp chèn như: Hình 14

```
Nhập số phần tử của mảng: 5
Nhập giá trị: 1
Nhập giá trị: 4
Nhập giá trị: 0
Nhập giá trị: 2
Nhập giá trị: -5
Mảng chưa sắp xếp:  [1, 4, 0, 2, -5]
Mảng đã sắp xếp: [-5, 0, 1, 2, 4]
```

Hình 14: Kết quả sắp xếp chèn

5.3 Sắp xếp nổi bọt

5.3.1 Giới thiệu về sắp xếp nổi bọt

Sắp xếp nổi bọt là một giải thuật sắp xếp đơn giản. Giải thuật sắp xếp này được tiến hành dựa trên việc so sánh cặp phần tử liền kề nhau và tráo đổi thứ tự nếu chúng không theo thứ tự.

Giải thuật này không thích hợp sử dụng với các tập dữ liệu lớn khi mà độ phức tạp trường hợp xấu nhất và trường hợp trung bình là $O(n^2)$ với n là số phần tử.

5.3.2 Cài đặt sắp xếp nổi bọt

Dưới đây là mã nguồn sắp xếp nổi bọt (tăng dần) như: Bảng 18.

def sap_xep_noi(lst):
  lst_1=lst.copy()
  for i in range(len(lst)-1):
  for j in range(i+1,len(lst)):
  if lst_1[i]>lst_1[j]:
  lst_1[i],lst_1[j]=lst_1[j],lst_1[i]
  return lst_1
n=int(input('Nhập số phần tử của mảng: '))
lst=[]
for i in range (0,n,1):
  lst.append(int(input('Nhập giá trị: ')))
print('Mảng chưa sắp xếp: ', lst)
print('Mản đã sắp xếp:',sap_xep_noi(lst))
	- Dựng hàm sap_xep_noi với tham số lst là 1 list số thực.
	- Vòng lặp ngoài lặp từ phần tử đầu tiên đến phần từ kế cuối của danh sách.
	- Vòng lặp trong lặp từ vị trí của phần tử vòng lặp phía ngoài đến cuối danh sách.
	- Mỗi vòng lặp như vậy sẽ so sánh từng đôi một nếu phần tử phía trước lớn hơn phần tử phía sau thì hoán đổi vị trí cho nhau.

Bảng 18: Mã nguồn sắp xếp nổi bọt

Dưới đây là kết quả sắp xếp nổi bọt như: Hình 15.

```
Nhập số phần tử của mảng: 5
Nhập giá trị: 2
Nhập giá trị: 3
Nhập giá trị: 1
Nhập giá trị: 0
Nhập giá trị: 8
Mảng chưa sắp xếp:  [2, 3, 1, 0, 8]
Mản đã sắp xếp: [0, 1, 2, 3, 8]
```

Hình 15: Kết quả sắp xếp nổi bọt

5.4 Sắp xếp nhanh

5.4.1 Giới thiệu sắp xếp nhanh

Sắp xếp nhanh còn gọi là Quick Sort là một kiểu sắp xếp phân chia, dựa trên kiểu phân mảng dữ liệu thành các nhóm phần tử nhỏ hơn.

Giải thuật sắp xếp nhanh chia mảng thành hai phần bằng cách so sánh với phần tử chốt. Một mảng gồm các phần tử nhỏ hơn phần tử chốt và một mảng các phần tử lớn hơn phần tử chốt. Quá trình phân chia này xảy ra cho đến khi độ dài của các mảng con đều bằng 1. Với phương pháp đệ quy ta có thể sắp xếp nhanh các mảng con thành mảng hoàn chỉnh.

5.4.2 Cài đặt sắp xếp nhanh

Dưới đây là mã nguồn cài đặt sắp xếp nhanh như: Bảng 19.

n=int(input('Nhập số phần tử của mảng: '))
lst=[]
for i in range (0,n,1):
  lst.append(int(input('Nhập giá trị: ')))
print('Mảng chưa sắp xếp: ', lst)
 	
def partition(arr, low, high):
  pivot = arr[high]
  i = low - 1
  for j in range(low, high):
  if arr[j] <= pivot:
  i += 1
  arr[i], arr[j] = arr[j], arr[i]
  arr[i + 1], arr[high] = arr[high], arr[i + 1]
  return i + 1
	- Hàm partition dùng để chia nhỏ và sắp xếp mảng.
	- Biến pivot dùng để lưu phần tử chốt, low là chỉ số bắt đầu của mảng con.
	- Nếu phần tử hiện tại bé hơn phần tử chốt thì chỉ số bắt đầu mảng con tăng lên 1 và hoán đổi hai vị trí đó.
	
	- Hàm quicksort sử dụng giải thuật đệ quy để giải quyết

def quicksort(arr, low, high):
  if low < high:
  pi = partition(arr, low, high)
  quicksort(arr, low, pi - 1)
  quicksort(arr, pi + 1, high)
quicksort(arr, 0, len(arr) - 1)
print('Mản đã sắp xếp:',(lst))	

Bảng 19: Mã nguồn cài đặt Quick sort

Dưới đây là kết quả của Quick sort như: Hình 16.

```
Nhập số phần tử của mảng: 5
Nhập giá trị: 2
Nhập giá trị: 0
Nhập giá trị: 8
Nhập giá trị: 1
Nhập giá trị: 6
Mảng chưa sắp xếp:  [2, 0, 8, 1, 6]
Mảng đã sắp xếp: [0, 1, 2, 6, 8]
```

Hình 16: Kết quả Quick sort

5.5 Sắp xếp vun đống (Heap sort)

5.5.1 Giới thiệu về sắp xếp vun đống

Sắp xếp vun đống – heap sort là một thuật toán sắp xếp nhanh sử dụng kĩ thuật phân loại dựa trên cấu trúc cây nhị phân đặc biệt gọi là đống nhị phân (binary heap). Thuật toán dựa vào sự đặc biệt của cây nhị phân để lựa chọn ra phần tử lớn nhất rồi lần lượt chèn phần tử này vào vùng sắp xếp.

5.5.2 Cài đặt sắp xếp vun đống

Dưới đây là mã nguồn sắp xếp vun đống như: Bảng 20

def heapify(arr, n, i):
  largest = i
  left = 2 * i + 1
  right = 2 * i + 2
 	
  if left < n and arr[i] < arr[left]:
  largest = left
	- Hàm heapity dùng để duy trì tính vun đống của mảng cần sắp xếp.
	- Tham số arr chứa mảng, n là số lượng phần tử cần xét, i là chỉ số nút gốc hiện tại vun đống.

if right < n and arr[largest] < arr[right]:
  largest = right
 	
if largest != i:
  arr[i], arr[largest] = arr[largest], arr[i]
  heapify(arr, n, largest)
 	
def heap_sort(arr):
  n = len(arr)
  for i in range(n // 2 - 1, -1, -1):
  heapify(arr, n, i)
 	
for i in range(n - 1, 0, -1):
  arr[i], arr[0] = arr[0], arr[i]
  heapify(arr, i, 0)
	- So sánh nút gốc với nút con bên trái, nếu thỏa gán chỉ số của nút con bên trái cho biến largest.
	
	- So sánh nút gốc với nút con bên phải, nếu thỏa mản gán lại chỉ số cho biến largest.
	
	- Nếu nút gốc không lớn nhất, hoán đổi với nút con lớn nhất và tiếp tục đệ quy.
	
	- Hàm heap_sort dùng để thực hiện thuật toán sắp xếp, n là độ dài của mảng cần sắp xếp.
	- Xây dựng một vun đống (heap) từ mảng arr bằng cách duyệt qua các nút gốc từ phần tử cuối cùng có nút con lớn nhất (có chỉ số (n // 2) - 1) lên đầu mảng. Đối với mỗi nút gốc, gọi hàm heapify(arr, n, i) để tạo và duy trì tính chất vun đống.

Bảng 20: Mã nguồn cài đặt Heap sort

Dưới đây là kết quả sắp xếp vun đống như: Hình 17.

```
Nhập số phần tử của mảng: 5
Nhập giá trị: 2
Nhập giá trị: 0
Nhập giá trị: 1
Nhập giá trị: 6
Nhập giá trị: 7
Mảng chưa sắp xếp:  [2, 0, 1, 6, 7]
Mảng đã sắp xếp: [0, 1, 2, 6, 7]
```

Hình 17: Kết quả sắp xếp vun đống

5.6 Sắp xếp trộn (Merge sort)

5.6.1 Giới thiệu sắp xếp trộn

Merge sort hoạt động trên nguyên tắc chia để trị. Merge sort chia nhỏ nhiều lần một mảng thành hai mảng con bằng nhau cho đến khi mỗi mảng con bao gồm một phần tử duy nhất. Cuối cùng, tất cả các mảng con đó được hợp nhất để sắp xếp mảng kết quả.

Ý tưởng của giải thuật này bắt nguồn từ việc trộn hai danh sách đã sắp xếp thành một danh sách mới cũng được sắp xếp.

5.6.2 Cài đặt sắp xếp trộn

def merge_sort(arr):
  if len(arr) <= 1:
  return arr
 	
  mid = len(arr) // 2
  left_half = arr[:mid]
  right_half = arr[mid:]
 	
  left_half = merge_sort(left_half)
  right_half = merge_sort(right_half)
 	
  return merge(left_half, right_half)
 	
def merge(left, right):
  merged = []
  left_index = 0
  right_index = 0
 	
  while left_index< len(left) and right_index<len(right):
  if left[left_index]< right[right_index]:
  merged.append(left[left.index])
  left_index+=1
  else:
  merged.append(right[right_index])
  right_index += 1
 	
  while left_index < len(left):
  merged.append(left[left_index])
  left_index += 1
 	
  while right_index < len(right):
  merged.append(right[right_index])
	- Khởi tạo hàm merge_sort
	- Nếu mảng chỉ một phần tử thì trả về mảng đó không cần sắp xếp.
	- Chia mảng thành hai phần bằng cách tìm điểm giữa
	
	- Đệ quy gọi hàm merge_sort trên từng nửa mảng
	- Gộp hai nửa mảng đã sắp xếp để tạo mảng mới.
	
	- Tạo hàm merge
	- biến merged dùng để chứa mảng cuối cùng đã sắp xếp.
	
	- So sánh các phần tử từ hai nửa mảng và đưa vào mảng merged
	
	- Đưa các phần tử còn lại của left vào merged
	
	- Đưa các phần tử còn lại của right vào merged

```
        right_index += 1

    return merged


n=int(input('Nhập số phần tử của mảng: '))
lst=[]
for i in range (0,n,1):
  lst.append(int(input('Nhập giá trị: ')))
print('Mảng chưa sắp xếp: ', lst)
print('Mảng đã sắp xếp:',merge_sort(lst))
```

Bảng 21: Mã nguồn cài đặt Merge sort

Dưới đây là kết quả sau khi thực hiện chương trình sắp xếp trộn như: Hình 18.

```
Nhập số phần tử của mảng: 6
Nhập giá trị: 1
Nhập giá trị: 0
Nhập giá trị: 2
Nhập giá trị: -3
Nhập giá trị: 4
Nhập giá trị: -2
Mảng chưa sắp xếp:  [1, 0, 2, -3, 4, -2]
Mảng đã sắp xếp: [-3, -2, 0, 1, 2, 4]
```

Hình 18: Kết quả thuật toán sắp xếp trộn