from \_\_future\_\_ import annotations  
from typing import TypeVar, Iterable, Sequence, Generic, \  
 List, Callable, Set, Deque, Dict, Any, Optional  
from typing\_extensions import Protocol  
from heapq import heappush, heappop  
  
  
T = TypeVar('T')  
  
  
def linear\_contains(iterable: Iterable[T], key: T) -> bool:  
 for item in iterable:  
 if item == key:  
 return True  
 return False  
# end of function linear\_contains  
  
  
C = TypeVar("C", bound="Comparable")  
  
  
# Q 01. Protocol ? - Nominal and Structural Subtyping  
# Q 02. 위 예제에서 한 가지 불편한 점은 타입 힌트를 위해  
# Comparable 클래스를 구현해야 한다는 것이다.  
# Comparable 타입은 비교 연산자 (<, >, =등)를 구현하는 타입이다.  
# 파이썬 다음 버전에서는 이런한 공통 연산자를 구현하는 타입에 대한  
# 타입 힌트를 작성하는 보다 간결한 방법이 있어야 할 것이다.  
# >> 현재 버전에서 아래의 사항들을 정의해주지 않아도 문제 없는데...?  
class Comparable(Protocol):  
 def \_\_eq\_\_(self, other: Any) -> bool:  
 ...  
  
 def \_\_lt\_\_(self: C, other: C) -> bool:  
 ...  
  
 def \_\_gt\_\_(self: C, other: C) -> bool:  
 return (not self < other) and self != other  
  
 def \_\_le\_\_(self: C, other: C) -> bool:  
 return self < other or self == other  
  
 def \_\_ge\_\_(self: C, other: C) -> bool:  
 return not self < other  
# end of class Comparable  
  
  
def binary\_contains(sequence: Sequence[C], key: C) -> bool:  
 low: int = 0  
 high: int = len(sequence) - 1  
 while low <= high:  
 mid: int = (low + high) // 2  
 if sequence[mid] < key:  
 low = mid + 1  
 elif sequence[mid] > key:  
 high = mid - 1  
 else:  
 return True  
 return False  
# end of function binary\_contains  
  
  
# Q 03. @property 의미?  
# A 03. get, set에 대한 느낌  
class Stack(Generic[T]):  
 def \_\_init\_\_(self) -> None:  
 self.\_container: List[T] = []  
  
 @property  
 def empty(self) -> bool:  
 return not self.\_container  
  
 def push(self, item: T) -> None:  
 self.\_container.append(item)  
  
 def pop(self) -> T:  
 return self.\_container.pop()  
  
 def \_\_repr\_\_(self) -> str:  
 return repr(self.\_container)  
# end of Stack  
  
  
class Queue(Generic[T]):  
 def \_\_init\_\_(self) -> None:  
 # Error?  
 self.\_container: Deque[T] = Deque()  
  
 @property  
 def empty(self) -> bool:  
 return not self.\_container  
  
 def push(self, item: T) -> None:  
 self.\_container.append(item)  
  
 def pop(self) -> T:  
 return self.\_container.popleft()  
  
 def \_\_repr\_\_(self) -> str:  
 return repr(self.\_container)  
# end of Queue  
  
  
class PriorityQueue(Generic[T]):  
 def \_\_init\_\_(self) -> None:  
 self.\_container: List[T] = []  
  
 @property  
 def empty(self) -> bool:  
 return not self.\_container  
  
 def push(self, item: T) -> None:  
 heappush(self.\_container, item)  
  
 def pop(self) -> T:  
 return heappop(self.\_container)  
  
 def \_\_repr\_\_(self) -> str:  
 return repr(self.\_container)  
# end of PriorityQueue  
  
  
# Q 04. Optional 의미?  
# TIP : Optional 매개변수는 매개변수가 있다면 해당 타입의 값이 변수에 의해 참조되거나  
# None이 참조 될 수 있음을 의미한다.  
# None이 어떻게 참조되지?  
# A 04. 초기화를 그렇게 하겠다.  
# frontier.push(Node(initial, None))  
# frontier.push(Node(child, current\_node))  
class Node(Generic[T]):  
 def \_\_init\_\_(self, state: T, parent: Optional[Node],  
 cost: float = 0.0, heuristic: float = 0.0) -> None:  
 self.state: T = state  
 self.parent: Optional[Node] = parent  
 self.cost: float = cost  
 self.heuristic: float = heuristic  
  
 def \_\_lt\_\_(self, other: Node) -> bool:  
 return (self.cost + self.heuristic) < (other.cost + other.heuristic)  
# end of Node  
  
  
def dfs(initial: T, goal\_test: Callable[[T], bool],  
 successors: Callable[[T], List[T]]) -> Optional[Node[T]]:  
 frontier: Stack[Node[T]] = Stack()  
 frontier.push(Node(initial, None))  
 # 아마도 중복 방지  
 explored: Set[T] = {initial}  
  
 while not frontier.empty:  
 current\_node: Node[T] = frontier.pop()  
 current\_state: T = current\_node.state  
  
 # goal\_test를 그냥 이렇게 불러올 수 있나?  
 # -> 객체 만들고 전달  
 if goal\_test(current\_state):  
 return current\_node  
  
 for child in successors(current\_state):  
 if child in explored:  
 continue  
  
 explored.add(child)  
 frontier.push(Node(child, current\_node))  
 return None  
# end of dfs  
  
  
def bfs(initial: T, goal\_test: Callable[[T], bool],  
 successors: Callable[[T], List]) -> Optional[Node[T]]:  
 frontier: Queue[Node[T]] = Queue()  
 frontier.push((Node(initial, None)))  
 explored: Set[T] = {initial}  
  
 while not frontier.empty:  
 current\_node: Node[T] = frontier.pop()  
 current\_state: T = current\_node.state  
  
 if goal\_test(current\_state):  
 return current\_node  
  
 for child in successors(current\_state):  
 if child in explored:  
 continue  
  
 explored.add(child)  
 frontier.push(Node(child, current\_node))  
 return None  
# end of bfs  
  
  
def astar(initial: T, goal\_test: Callable[[T], bool],  
 successors: Callable[[T], List[T]], heuristic: Callable[[T], float]) -> Optional[Node[T]]:  
 frontier: PriorityQueue[Node[T]] = PriorityQueue()  
 frontier.push(Node(initial, None, 0.0, heuristic(initial)))  
 explored: Dict[T, float] = {initial: 0.0}  
  
 while not frontier.empty:  
 current\_node: Node[T] = frontier.pop()  
 current\_state: T = current\_node.state  
  
 if goal\_test(current\_state):  
 return current\_node  
  
 for child in successors(current\_state):  
 new\_cost: float = current\_node.cost + 1  
  
 if child not in explored or explored[child] > new\_cost:  
 explored[child] = new\_cost  
 frontier.push(Node(child, current\_node, new\_cost, heuristic(initial)))  
 return None  
# end of astar  
  
  
def node\_to\_path(node: Node[T]) -> List[T]:  
 path: List[T] = [node.state]  
 while node.parent is not None:  
 node = node.parent  
 path.append(node.state)  
 path.reverse()  
 return path  
# end of note\_to\_path  
  
# if \_\_name\_\_ == "\_\_main\_\_":  
# print(linear\_contains([1, 5, 15, 15, 15, 15, 20], 5))  
# print(binary\_contains(["a", "d", "e", "f", "z"], "f"))  
# print(binary\_contains(["john", "mark", "ronald", "sarah"], "sheila"))