```python
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, set 에 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
# None 이 참조 될 수 있음을 의미한다.
# None 이 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_test 를 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
```

```python
# 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"))
```