```
import heapq

class Graph:
    def _init_(self):
        self.nodes = {}

    def add_node(self, name, heuristic):
        self.nodes[name] = {'heuristic': heuristic, 'edges': {}}

    def add_edge(self, from_node, to_node, cost):
        self.nodes[from_node]['edges'][to_node] = cost

    def best_first_search(self, start, goal):
        priority_queue = []
        heapq.heappush(priority_queue, (self.nodes[start]['heuristic'], start))
        came_from = {}
        explored = set()

        while priority_queue:
            current_heuristic, current_node = heapq.heappop(priority_queue)

            if current_node in explored:
                continue

            explored.add(current_node)

            if current_node == goal:
                return self.reconstruct_path(came_from, current_node)

            for neighbor in self.nodes[current_node]['edges']:
                if neighbor not in explored:
```

```
                came_from[neighbor] = current_node
                heapq.heappush(priority_queue, (self.nodes[neighbor]['heuristic'], neighbor))

        return None  # No path found

    def reconstruct_path(self, came_from, current):
        total_path = [current]
        while current in came_from:
            current = came_from[current]
            total_path.append(current)
        return total_path[::-1]  # Reverse the path

def main():
    graph = Graph()
    graph.add_node("A", 10)
    graph.add_node("B", 5)
    graph.add_node("C", 2)
    graph.add_node("D", 0)

    graph.add_edge("A", "B", 1)
    graph.add_edge("A", "C", 4)
    graph.add_edge("B", "D", 1)
    graph.add_edge("C", "D", 2)

    start_node = 'A'
    goal_node = 'D'

    path = graph.best_first_search(start_node, goal_node)

    if path:
        print("Path found:", " -> ".join(path))
```

```
    else:
        print("No path found.")

if _name_ == "_main_":
    main()
```