11. More data structures

In this chapter we will learn about two data structures:

- *Deque*: Deque is a list structure that supports efficient addition and removal of elements both at the beginning and at the end of the list.
- *Heap*: Heap is a data structure where the smallest or the largest element can be found and removed efficiently.

Both data structures are available in the Python standard library and are thus easy to use in Python programming.

Deque

With the standard Python list, we can efficiently add an element to the end of a list using the method `append` and remove an element from the end of the list using the method `pop`. However, there is no efficient way of adding or removing an element at the beginning of the list.

Deque is a list structure where additions and removals at both ends of the list are efficient. In Python, deque is available in the module `collections` and has the following methods:

- `append`: add an element to the end of the list
- `pop`: remove the element at the end of the list
- `appendleft`: add an element to the beginning of the list
- `popleft`: remove the element at the beginning of the list

In addition to the familiar methods `append` and `pop`, we now have the methods `appendleft` and `popleft` that operate at the beginning of the list. The time complexity of all four methods is $O(1)$.

The following code illustrates the use of a deque:

```
import collections

items = collections.deque()
```

```python
items.append(1)
items.append(2)
items.appendleft(3)
items.append(4)
items.appendleft(5)

print(items) # deque([5, 3, 1, 2, 4])

print(items[0]) # 5
print(items[1]) # 3
print(items[-1]) # 4
```

As the code shows, the elements of a deque can be accessed using the indexing operator `[]` similarly to the standard list. However, an access to an element through an index can be inefficient in a deque, which is a major weakness of the deque. In the standard list any element can be accessed in $O(1)$ time, but an access to an element in the middle of a deque takes $O(n)$ time.

In Python, a deque is implemented using a linked list, which enables the efficient additions and removals at both ends.

Stack and queue

Stack is a data structure that supports adding and removing elements at the end of a list. *Queue* is a data structure that supports adding elements to end of a list and removing elements from the beginning of the list.

We can use a deque to implement both a stack and a queue, since operations at both ends are efficient. The following classes implement the stack and the queue:

```python
import collections

class Stack:
    def __init__(self):
        self.stack = collections.deque()

    def push(self, x):
        self.stack.append(x)

    def top(self):
        return self.stack[-1]
    def pop(self):
        self.stack.pop()

class Queue:
    def __init__(self):
        self.queue = collections.deque()

    def push(self, x):
        self.queue.append(x)

    def top(self):
        return self.queue[0]

    def pop(self):
        self.queue.popleft()
```

Notice that we could use the standard list to implement a stack as we did in Chapter 6. However, the implementation of a queue benefits from the efficient operations at both ends of a deque.

Heap

Heap is a data structure that supports adding, accessing and removing elements. Additions to a heap are unrestricted, but depending on the implementation of a heap, either only the smallest or only the largest element of the heap can be accessed and removed.

In Python, a list can be used as a heap with the following functions in the module `heapq`:

- `heappush`: add an element to the heap
- `heappop`: remove and return the smallest element of the heap

The time complexity of both functions is $O(\log n)$. In addition, the smallest element in the list is always at the beginning of the list and can be accessed in $O(1)$ time.

The following code illustrates using a heap in Python:

```
import heapq

items = []
```

```
heapq.heappush(items, 4)
heapq.heappush(items, 2)
heapq.heappush(items, 3)
heapq.heappush(items, 1)
heapq.heappush(items, 5)

print(items[0]) # 1
heapq.heappop(items)
print(items[0]) # 2
```

Compared to hashing, the advantage of a heap is the efficient access and removal of the smallest or the largest element. With hashing this is not possible because a hash table does not store the elements in order. On the other hand, a heap does not support access to any other element than the smallest or the largest.

Notice that a heap may contain multiple copies of an element. For example, the following code adds three copies of the number 1 into a heap:

```
import heapq

items = []
heapq.heappush(items, 1)
heapq.heappush(items, 1)
heapq.heappush(items, 1)

print(items) # [1, 1, 1]
```

The functions in the module `heapq` keep the elements of a list in a specific order so that the operations can be implemented efficiently.

Example: Sliding window

Task

You are given a list of n numbers and a parameter k. For each *sliding window*, i.e., a sublist of k consecutive elements, from the left to right, find the smallest element in the sublist.

For example, when the list is $[1, 2, 3, 5, 4, 4, 1, 2]$ and $k = 3$, the desired answer is $[1, 2, 3, 4, 1, 1]$. Here the sublists are $[1, 2, 3]$, $[2, 3, 5]$, $[3, 5, 4]$, $[5, 4, 4]$, $[4, 4, 1]$ and $[4, 1, 2]$.

We can solve the task efficiently using a heap as follows:

```
import heapq

def find_minima(items, k):
    n = len(items)
    heap = []
    result = []

    for i in range(n):
        heapq.heappush(heap, (items[i], i))
        while heap[0][1] <= i - k:
            heapq.heappop(heap)
        if i >= k - 1:
            result.append(heap[0][0])

    return result
```

The algorithm iterates through the list and adds pairs of form (x, i) to the heap, where x is the element at the position i. The smallest element in the heap can be accessed efficiently. If the smallest element is outside the current window, it is removed. Otherwise the smallest element is the desired answer for the current window.

The time complexity of the algorithm is $O(n \log n)$, because each element is added to and removed from the heap at most once.

Other programming languages

A deque can be implemented in different ways. The implementations in C++ and Java support efficient indexing of any element. In C++, the class `std::deque` implements the deque:

```
std::deque<int> items;

items.push_back(1);
items.push_back(2);
items.push_front(3);
items.push_back(4);
items.push_front(5);
```

Java has the class `ArrayDeque` based on an array:

```
ArrayDeque<Integer> items = new ArrayDeque<>();

items.addLast(1);
items.addLast(2);
items.addFirst(3);
items.addLast(4);
items.addFirst(5);
```

In many programming languages, heap is implemented under the name *priority queue*. In C++, the class `std::priority_queue` implements a max heap:

```
std::priority_queue<int> items;

items.push(1);
items.push(2);
items.push(3);

cout << items.top() << "\n"; // 3
items.pop();
cout << items.top() << "\n"; // 2
```

In Java, the class `PriorityQueue` implements a min heap:

```
PriorityQueue<Integer> items = new PriorityQueue<>();

items.add(1);
items.add(2);
items.add(3);

System.out.println(items.peek()); // 1
items.poll();
System.out.println(items.peek()); // 2
```

HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI