Mälardalens universitet

DVA245, Laboration 5 – Merge sort med köer

Mål
1- kunna använda abstrakta datatyper i programmeringsuppgifter
3- kunna implementera algoritmer utifrån beskrivningar i pseudokod
4- vara tillräckligt bekant med några vanliga algoritmer för sortering och sökning för att kunna implementera en tidigare okänd variant av algoritmen utifrån en informell beskrivning av förändringen

Merge sort baseras på följande idéer:

1. För att hitta det minsta elementet i två sorterade listor behöver vi bara jämföra de första elementen i båda listorna. Så när vi kombinerar/ mergar två listor till en behöver vi för varje element i den mergade listan bara jämföra två element – ett från varje in-lista.
2. En lista med bara ett element är sorterad.

En illustration av mergning av två sorterade listor finns i Fig. 1 på nästa sida.

Punkt ett betyder att när vi slår ihop listorna jämför vi första elementen i in-listorna för att se vilket av dessa två vi ska lägga till i slutet av den sorterade ut-listan. Det betyder att vi kan implementera merge sort med hjälp av köer, och det är vad ni ska göra i denna laboration.

I filen `merge_queue_begin.py` på Canvas finns kodskelett som testas i main-funktionen. I filen `merge_array.py` finns en rekursiv merge sort-implementation som använder listor som ni kan titta på och jämföra med.

Merge sort-implementation med köer

Ni ska implementera en imperativ (icke-rekursiv) variant av merge sort, som delas upp i tre funktioner:

1. `merge` som slår ihop två sorterade köer till en, som visas i fig. 1.
2. `merge_level_queues` som slår ihop köer två och två med hjälp av `merge` till hälften så många köer. Om antalet köer in är udda så flyttas den sista som den är till resultatet.
3. `merge_sort` som skapar en kö per element som ska sorteras (vi vet att köer med ett element är sorterade). Sedan anropas `merge_level_queues` gång på gång tills bara en sorterad kö finns kvar.

Hela processen illustreras i Fig. 2.

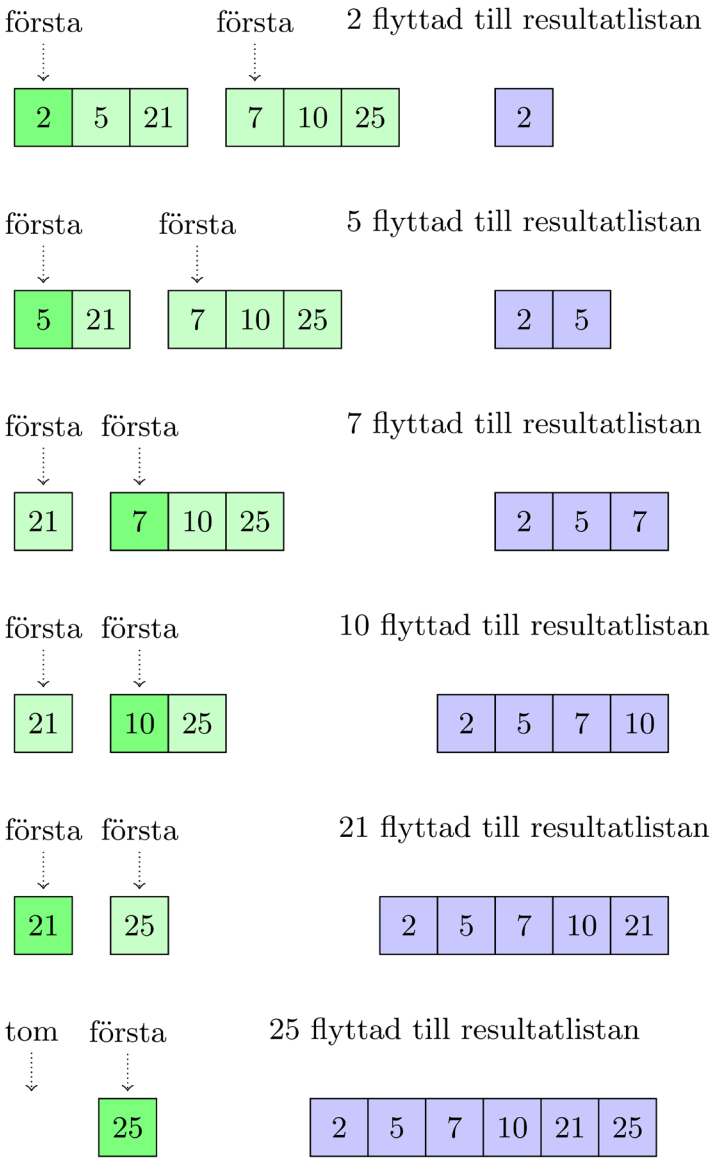

Fig 1: Sammanslagning (mergning) av två sorterade listor

Vi använder `deque` från pythons modul `collections` för köerna.
Dokumentation finns här:
https://docs.python.org/3/library/collections.html#collections.deque

För att lägga till element i köerna använder vi `deques append`, och för att ta bort använder vi `popleft`. Vi kan titta på det första elementet utan att ta bort det (top) med hjälp av operatorn `[0]`.

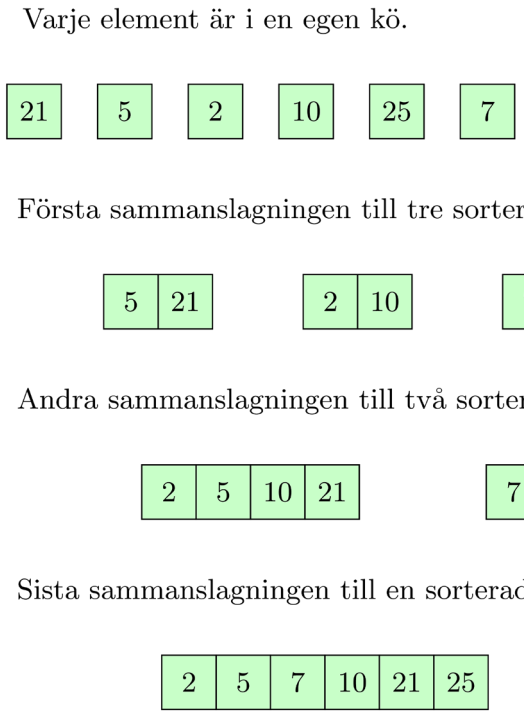

Fig. 2: Merge sort med köer

I filen `merge_queue_begin.py` finns kodskelett och i main-funktionen finns tester. Sätt er in i vad testerna gör och varför de inte går igenom inledningsvis. Ni kanske vill ta bort delar av testerna i början. I zip-arkivet finns också `merge_array.py` med en rekursiv merge sort-implementation för direktaccesslistor (pythonlistor). Implementationen med köer är inte rekursiv. Istället för att i varje rekursivt steg dela upp listan i två dellistor och anropa mergesort för varje halva tills vi når basfallet med en dellista som innehåller ett element, så börjar vi med att skapa sorterade köer med ett element i varje och sedan slå ihop dem två och två.

Merge

Implementera `merge`-funktionen som slår ihop två sorterade köer. Så länge båda in-köerna `S1` och `S1`innehåller element så jämför ni de första elementen i köerna. Ta bort det minsta av dem och lägg i resultatkön `S`. När en av in-köerna är tom flyttar ni de element som är kvar i den andra kön till resultatkön. Implementationen är ungefär 10 rader kod. I kodskelettet skapas och returneras resultatkön `S`, er implementation ska vara mellan dessa två rader.

Merge level queues

Implementera funktionen `merge_level_queues`. Den tar en kö med input-köer som visualiseras av en nivå/ rad i Fig. 2 och returnerar en kö med resultatköer som visualiseras av nivån/ raden under i samma figur. Pseudokod för `merge_level_queues` finns på nästa sida. I while-loopen tas inköer ut två

och två från `level_queues` och slås ihop. Resultatet läggs i `next_level_queues`. I if-satsen kontrolleras om det fanns ett udda antal köer i `level_queues`, i så fall läggs den sista kön till utan sammanslagning. Ni behöver inte följa pseudokoden, det går att lösa på andra sätt.

```
function MERGE_LEVEL_QUEUES(level_queues)
         returns a queue of merged queues
input:   level_queues – the queue of input queues
to merge
local variables:  next_level_queues – the queue of
merged queues
                  q1, q2 – input queues from
level_queues


next_level_queues ←the empty queue
while LEN(level_queues)> 1 do
         q1 ← level_queues.DEQUEUE()
         q2 ← level_queues.DEQUEUE()
         next_level_queues.ENQUEUE(MERGE(q1, q2))
if LEN(level_queues)> 0  then
         q1 ← level_queues.DEQUEUE()
         next_level_queues.ENQUEUE(q1)
return next_level_queues
```

Merge sort

Implementera `merge_sort` som tar en in- in-kö `S` och returnerar en kö med elementen i `S` sorterade.

1. Skapa nivå-kön.
2. För varje element i `S`, ta ut det och lägg in det som enda element i en ny kö. Se första raden/ nivån i Fig. 2.
3. Så länge nivå-kön innehåller mer än en sorterad kö, slå ihop köerna två och två i en nivå, och ersätt nivå-kön med resultatet från anropet till `merge_level_queues`. Varje varv i while-loopen motsvarar ett steg nedåt i Fig. 2.
4. När nivå-kön bara innehåller en sorterad kö, ta ut den och returnera den.

Totalt blir det mindre än 10 rader kod.

För redovisning:
Ni behöver kunna förklara er implementation och hur testerna visar att era funktioner fungerar.

English:

Learnng objectives
1- be able to use abstract data types in programming assignments
3- be able to implement algorithms given descriptions as pseudocode
4-be sufficiently familiar with some common algorithms for sorting and searching to be able to implement a thus far unkown variant of the algorithm based on an informal description of the change

Merge sort is an based on the following ideas:

1. To find the smallest element of two sorted lists, we only need to compare the first element of each list. So, when merging two sorted lists into one, for each element to move, we only need to compare two elements - one from each of the input lists.
2. A list with only one element is always sorted.

An illustration of a merge of two sorted lists is provided in Fig. 1.

Looking at the first point - when merging we compare the first elements of the input lists, to determine which of these two elements to add to the end of the sorted output list. From this it is clear that we can implement merge sort using queues, and this is what you will do in this lab.

In the file `merge_queue_begin.py` on Canvas there is a code skeleton that is tested in the main function. In `merge_array.py` there a recursive merge sort-implementation that uses lists that you can look at and compare with.

Merge sort-implementation with queues

You shall ska implement an imperative (non-recursive) version of merge sort, divided into three functions:

1. `merge`, merging two sorted queues into one, as illustrated in Fig. 1
2. `merge_level_queues`, merging a number of queues two-by-two using `merge` into half the number of merged queues. If the number of queues is odd the last one is added to the resulting queues without merging.
3. `merge_sort` , that creates a queue for each element to sort (since we know that a single element is sorted), and repeatedly calls `merge_level_queues` until just one sorted queue remains.

The full process is illustrated in Fig. 2.

We use the `deque` from python's `collections` module for our queues. See the documentation:
https://docs.python.org/3/library/collections.html#collections.deque.

To enqueue elements we use `append`, and to dequeue elements we use `popleft`. The front element of the queue can be accessed without dequeueing with the operator `[0]`.

The file `merge_queue_begin.py` contains code skeletons and in the main-function there are tests. Go through what the tests do and why they don't pass initially. Maybe you want to remove parts of the tests in the beginning. In the zip-archive there is also `merge_array.py` with a recursive merge sort implementation for arrays (python lists). The queue-based merge sort that you will implement is not recursive. Instead of dividing a list until we have the base case of one-element lists that are already sorted, we start by creating sorted queues containing just one element, and we start merging them two-by-two.

Merge

Implement the `merge` function to merge two sorted queues. As long as both input queues `S1` and `S2` contain elements, compare the front elements of both queues. Dequeue the smallest, and enqueue on the output queue `S`. When one of the input queues is empty, move the remaining elements from the non-empty input queue to the output queue. This requires about 10 lines of code. In the skeleton the queue `S` is created and returned, you need to add the described functionality in between.

Merge level queues

Implementera funktionen `merge_level_queues`. It takes a queue of input-queues visualized by one level/ row in Fig. 2 and returns a queue with output-queues visualized by the level/ row below in the same figure. Pseudo code for `merge_level_queues` is in the Swedish section. In the while-loop, in-queues are dequeued two-by-two from `level_queues` and merged. The result is enqueued on `next_level_queues`. In the if-clause it is checked if there is an odd number of queues in `level_queues`, in that case the last queue is added without merging. You don't need to follow the pseudo code, it can be solved differently.

Merge sort

Implement the `merge_sort` function, that takes an input queue `S` and returns a queue with the elements of `S` sorted.

1. Create the level queue.

2. For each element in `S`, dequeue it and add it as the single element of a new queue. Add the new single-element queue to the level queue. See the first line of Fig. 2.
3. While the level queue contains more than one sorted queue, merge one level, and replace the level queue with the queue returned from `merge_level_queues`. For each iteration of the while-loop, we move one step down in Fig. 2.
4. When there is just one sorted queue remaining in the level queue, dequeue it and return the resulting sorted queue.

The function contains less than 10 lines of code.

For presentation:
You need to be able to explain the implementation and how the tests show that it works.