Data Structures and Algorithms

Dr. L. Rajya Lakshmi

Stack

- A container of data elements/objects that are stored and removed according to last in first out (LIFO) principle
- Objects are inserted at the top of stack and also removed from the top, that is, can remove the most recently inserted object at a time
- Insertion: pushing; deletion: popping

Stack

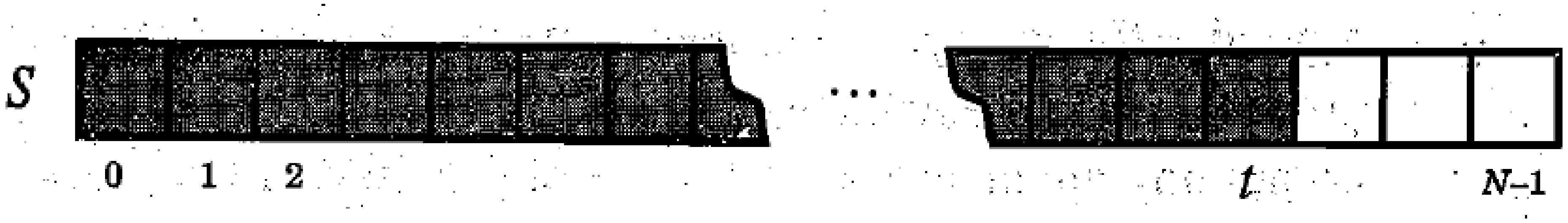
- Stack is an abstract data type that supports these operations:
  - push(e): inserts the object “e” at the top of the stack
  - pop(): removes and returns the top object from stack, if the stack is empty an error occurs

Stack

- The supporting operations of the stack ADT are:
  - size() – returns the number of elements in the stack
  - isEmpty() – Tells us whether there are any objects in the stack or not
  - top() - returns the top object of the stack, without removing the object, if the stack is empty an error occurs

Implementation using an array

- A stack can be easily implemented with an N-element array S
- The elements are stored from S[0] to S[t], “t” is the index of the topmost element in the stack



Implementation using an array

- Assume that array index starts from “0”, and initialize t to “-1”
- The value of t is used to identify when the stack is empty, and the size of the stack
- An exception must be raised when the stack becomes full

Algorithm push(o)

```
    if size()  = N then
        indicate a stack-full error has occurred
    t ←t+1
    S[t] = o
```

Implementation using an array

Algorithm pop()

    if(isEmpty()) then

        indicate a stack-empty error has occurred

    e ← S[t] {e is a temporary variable}

    S[t] ← null

    t ← t-1

    return e

Implementation using an array

Algorithm push(e)

    if size() == N then

        A←new array of length f(N)

    for i ←0 to N-1

        A[i] ← S[i]

    S ←A

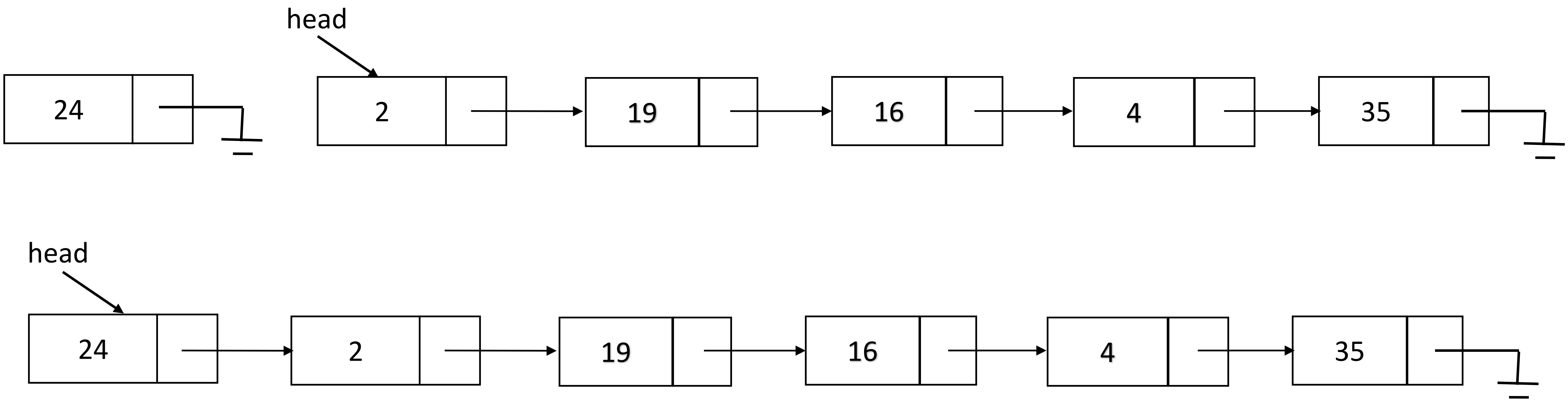
    t ←t+1

    S[t] ← e

- tight strategy: f(N) = N+c
- Growth strategy: f(N) = 2N

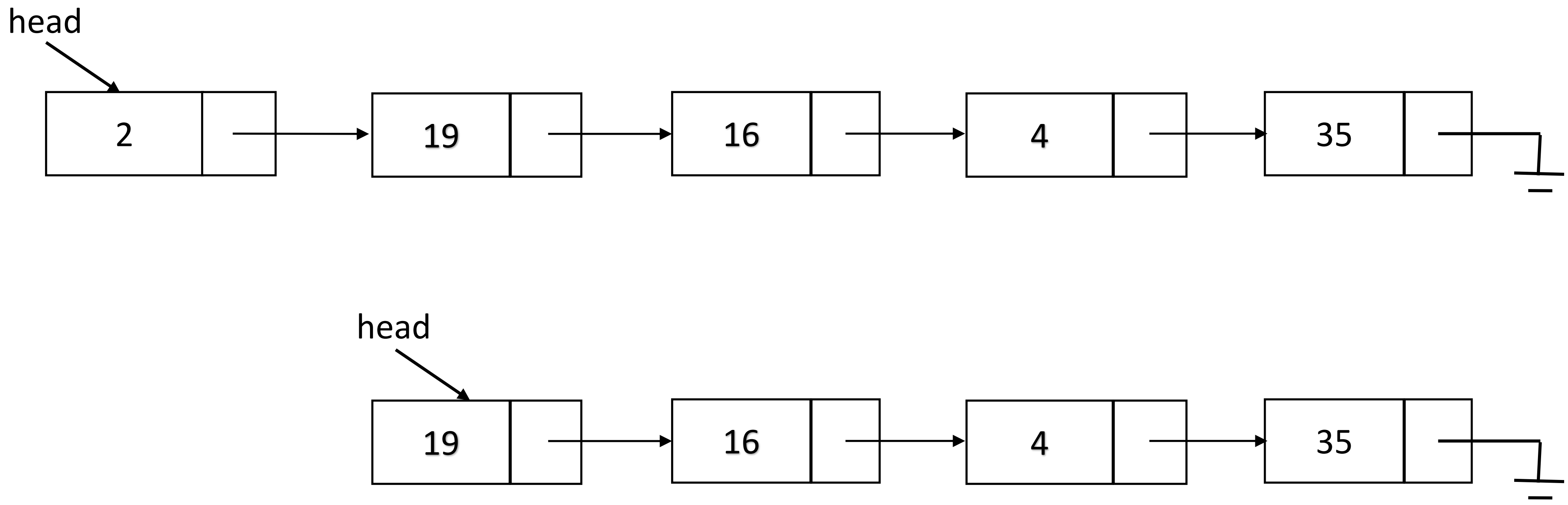
Implementation using a linked list

- push operation
  - Create a new node (temp) with the data element to be inserted
  - Update the next link of temp to point to the node referred by “head”
  - Update the head to refer to temp



Implementation using a linked list

- Pop operation
  - Update head to refer to next node of top of stack
  - Update the next link of top of stack to refer to null; free the memory allocated to deleted node



Implementation using a linked list

- Some of the methods implemented as a part of linked list ADT are:

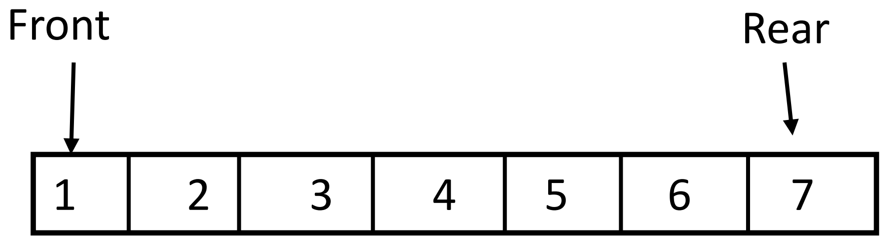
  first(), insertAtPosition(e,p), remove(p), retrieve(p)

Array or linked list based implementation

- Must assume a fixed upper bound on the ultimate size of the stack
- Waste of memory
- Linked lists:
- Do not have size limitation
- Use space in proportion to the number of elements in the stack

Queue

- A queue is container of data elements/objects that are inserted and removed according to the first-in-first-out (FIFO) principle
- Elements can be inserted at any point of time
- Can only remove the element which has been there for the longest
- Elements enter the queue at the “rear” and removed from the “front”



Queue: ADT

- Keeps objects in a sequence
- Access and deletion is limited to the first (front) element in the sequence
- Insertion is restricted to the end (rear) of the sequence

Fundamental methods:

- enqueue(o)
- dequeue()

Supporting methods:

- size()
- isEmpty()
- front()

Implementation using an array

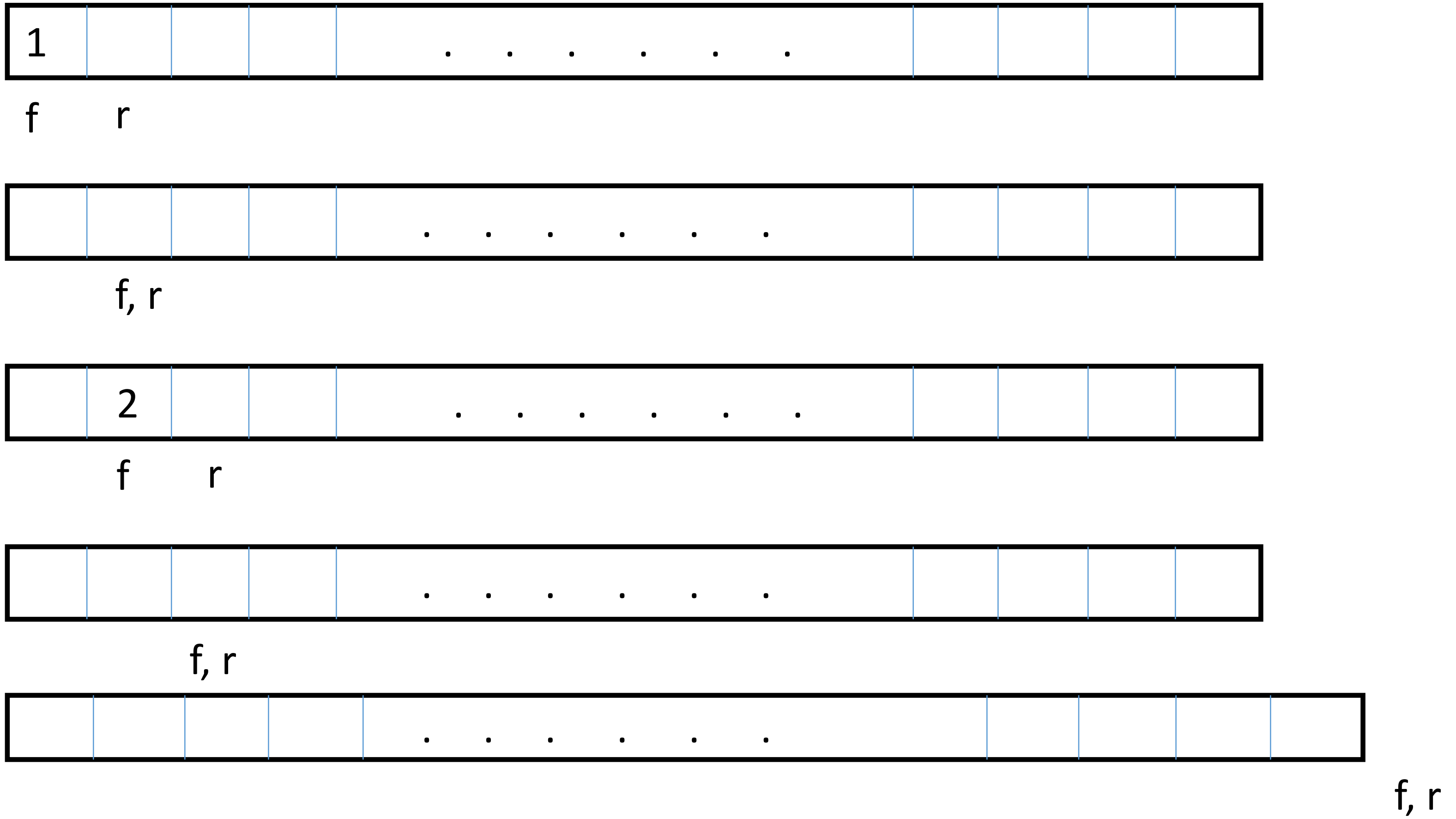
- Define an array Q of size N
- Define two variables “f” and “r” to enforce FIFO principle

  f: index to the cell of Q storing the first element of the queue

  r: index to the next available array cell of Q
- f = r = 0
- If f = r indicate that the queue is empty
- Increment f when an element is removed from the queue
- Increment r when an element is inserted into the queue

Implementation using an array

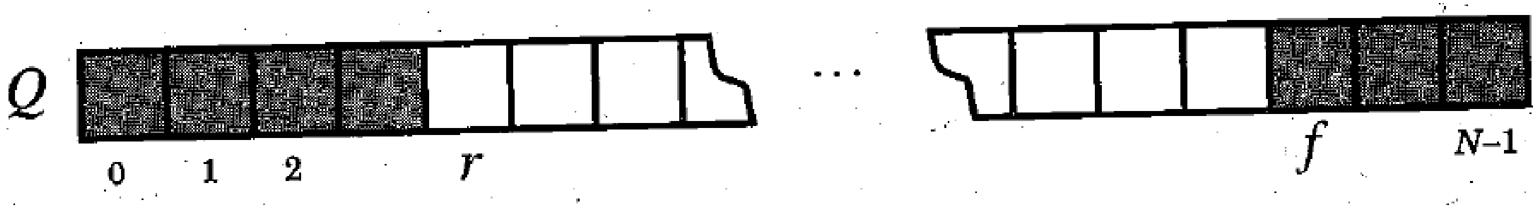
- Take an empty queue
- Insert an element and remove it; repeat this cycle for N times



Implementation using an array

- Insertion results in array-out-of-bounds error
- Not able to insert in spite of plenty empty cells
- Let "f" and "r" wrap around the queue
- View Q as a circular array
- f = (f+1) mod N; r = (r+1) mod N

(a)



(b)

Implementation using an array

- Consider the scenario, enqueue N elements one-by-one
- f=r
- Ambiguity in distinguishing between an empty and a full queue
- Do not let the queue to hold more than N-1 elements

Implementation using an array

Algorithm enqueue(e)

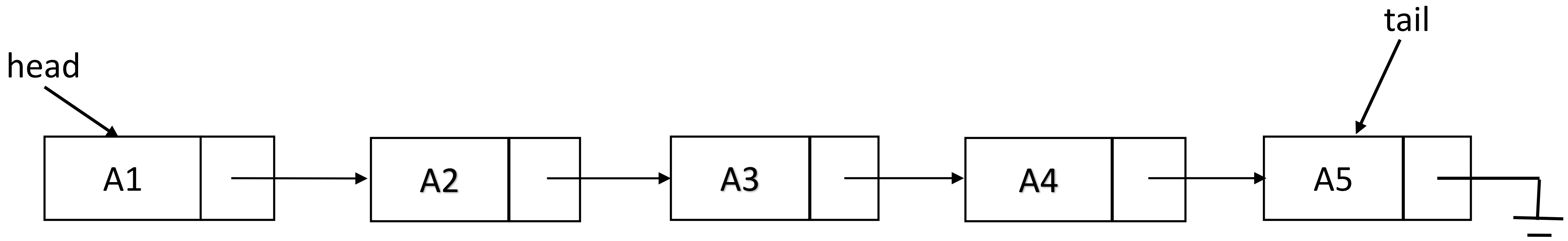
```
if (size() == N-1)
    raise QueueFull exception
Q[r] = e
r ← (r+1) mod N
```

Implementation using an array

```
Algorithm dequeue()
        if isEmpty() then
                raise QueueEmpty Exception
        temp ← Q[f]
        Q[f] = null
        f ← (f+1) mod N
        return temp
```

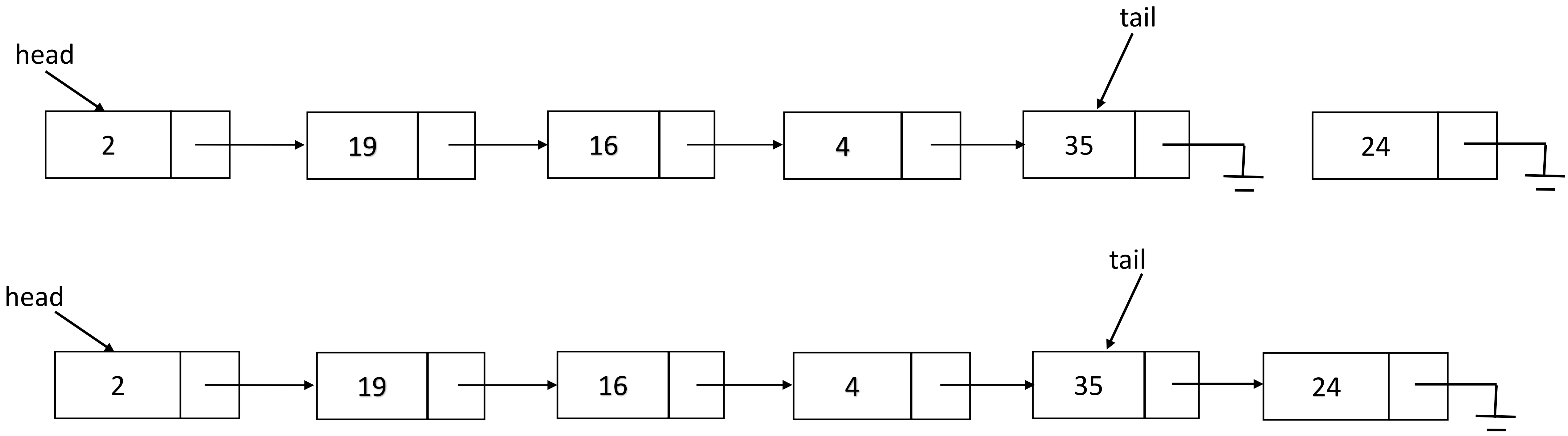
Implementation using a linked list

- What should be the front of the queue, head or tail?
- To reduce the overhead, head should be the front of the queue



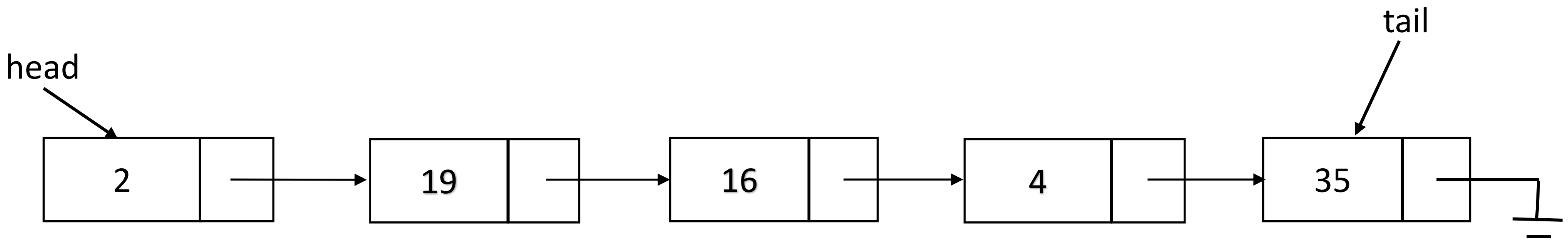
Implementation using a linked list

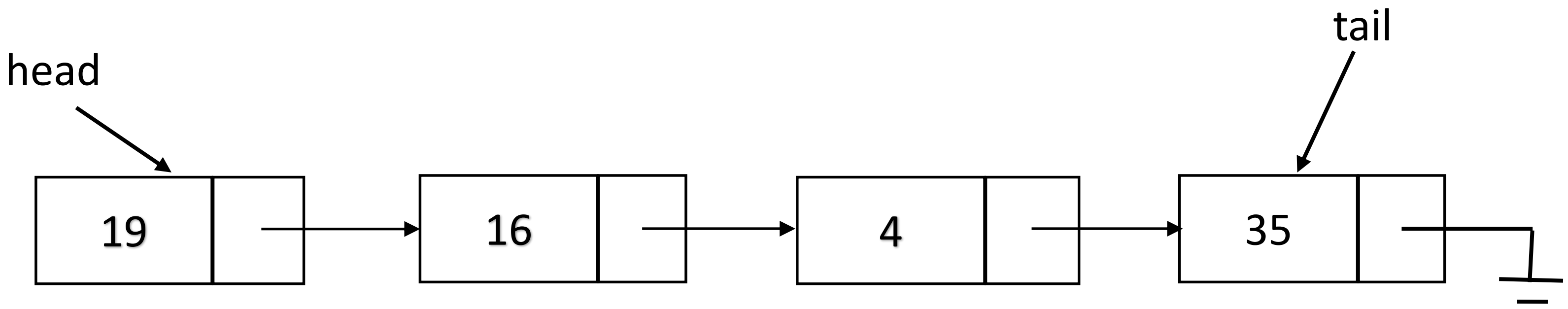
enqueue



Implementation using a linked list

- dequeue





Dynamic Sets

- Collection of objects whose size may change
- These collections of elements are called dynamic sets
- Lists, stacks and queues