CSCA48 Winter 2018
Week 3: Priority Queue, Linked Lists

Marzieh Ahmadzadeh, Nick Cheng

University of Toronto Scarborough

Administrative Detail

- Term test # 1 and #2 schedule is now on course website
- We will extend the deadline for the first assignment due to the closeness to your term test 1.

The Priority Queue ADT

- Requirement:
  - Every entry has a priority
  - `remove` operation, removes the entry with the highest priority
  - the priority is a positive integer, and the smaller the number is the higher is the priority
  - It is possible to have two entries with the same priority

- Application
  - Standby flyers
  - Auctions
  - Sorting a list

The Priority Queue ADT

- Data:
    - Any arbitrary objects/elements
- Operations:
    - Main:
        - insert(e,p): add element *e* with the priority of *p* to the PQ
        - extract_min(): remove and return the element with the highest priority
    - Auxiliary:
        - min(): returns the element with the highest rpiority
        - size(): returns the number of elements in the PQ
        - is_empty(): indicates whether or not the priority queue is empty
- Exception:
    - Raise EmptyPriorityQueueException if the PQ is empty and extract_min() or min() is requested

The PQ ADT Implementation

- Which one of these ADTs are suitable to implement a PQ?
  - dict, stack, queue, list?
- How many operation does it take to run the PQ methods if every access to elements counts as one operation?

Opearion	Unsorted List	Sorted List
size()	1	1
Is_empty()	1	1
insert(e,p)	1	n
min()	n	1
Extract_min()	n	1

- So we need a better ADT!
  - Don’t wait for it until week 6

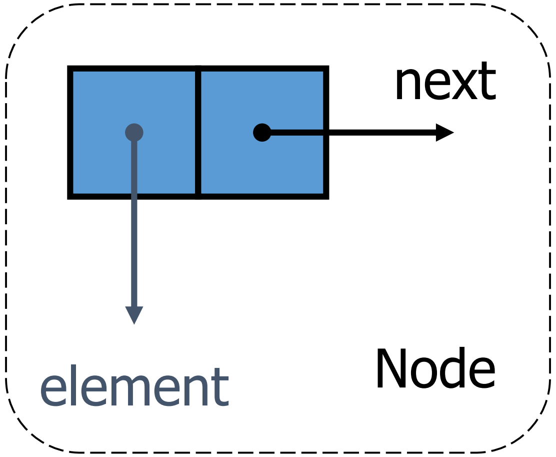
Lists

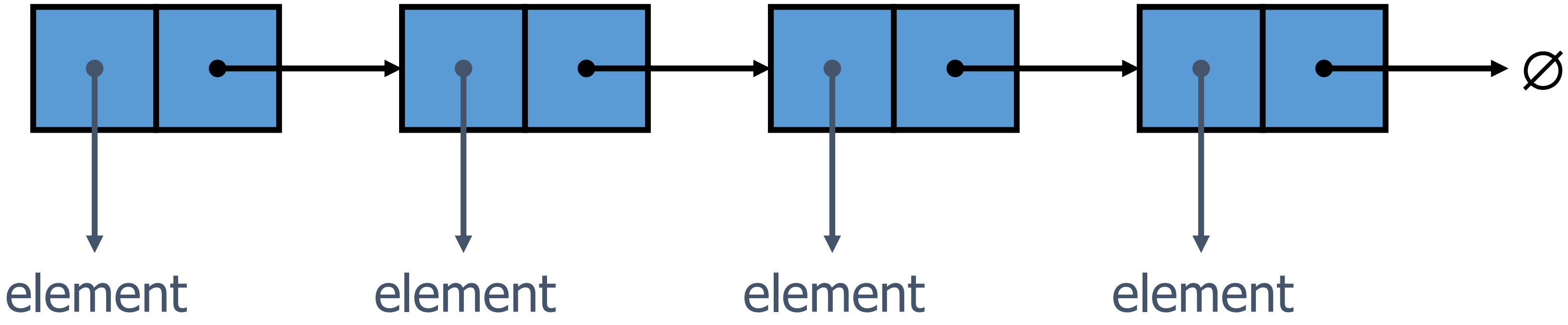
- You can insert any arbitrary data into a list.
- Elements are linearly accessed by their index.
- There is no limitation on the number of elements that can be added.

- List is an ADT itself.
  - So the question is how a list is implemented?
  - What is a concrete data structure behind implementing a list?

Single Linked Lists

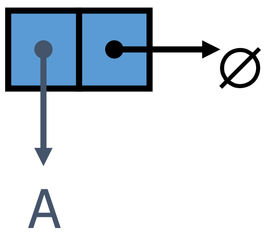
- A linked list is a concrete data structure, whose building block is a `Node`.
- Each node is an object that stores
  - a reference to an element
  - a reference called `next` to another node.
- The first Node is called the `head`
- The last node is called the `tail`
  - Tail has a None next reference .

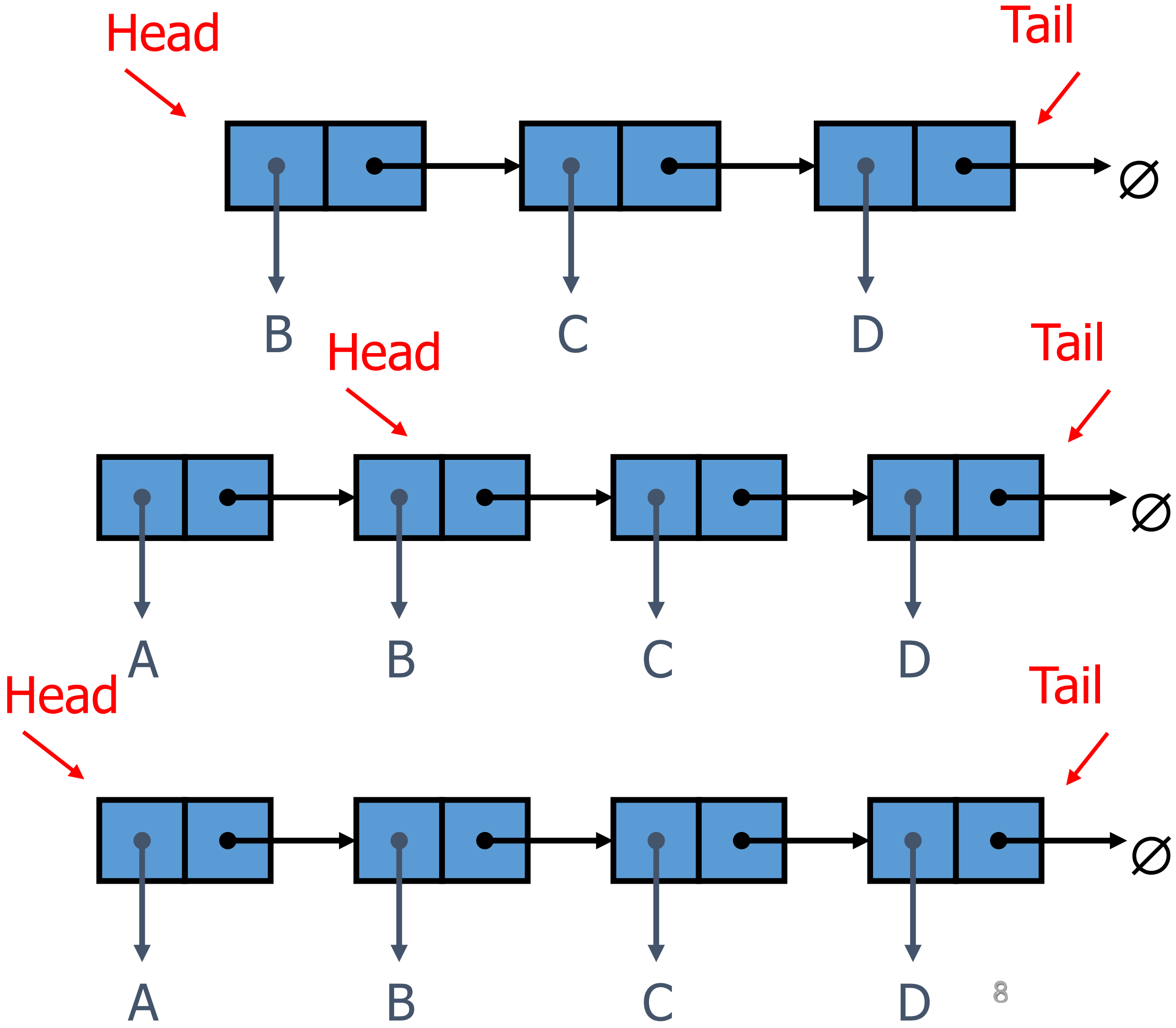




Inserting at the Head

1. Create a new node: Node(element, None)
2. Have the new node point to the old head
3. Update head to point to the new node
4. Add one to the size





Inserting at the Tail

1. Create a new node: Node(element, None)
2. Have tail to point to the new node
3. Update tail to point to the new node
4. Add one to the size

Remove from the head

1. Update head to point to next node in the list
2. Set the previous head to point to None
3. Decrement the size

- Don’t worry about the removed node, garbage collector deallocates the memory.

Removing from the tail

- Removing at the tail of a single linked list is not efficient!
- There is no way that we can update the tail to point to the previous node in a constant-time (i.e. operation)

Using SLL to implement other ADTs

- Having a concrete data structure such as linked lists in hand, you can implement other ADTs.
- How many operation does it take to run each ADTs method?
  - The Stack ADT
    - takes a constant time to push and pop (independent of the number of data in the satck)
  - The Queue ADT
    - takes a constant time to dequeue() and enqueue()
  - The List ADT
    - Takes a constant time to insert in each side
    - Takes a constant time to remove from the start of the list
    - Time required to remove from the end is dependent to the number of element in the list (i.e. n)
      - Not good! Need a better concrete data structure than single linked list!