COMP122/20 - Data Structures and Algorithms

06 Stacks and Queues

Instructor : Ke Wei 〖柯韋〗

➽ A319 ✆ Ext. 6452 ✉ wke@ipm.edu.mo

http://brouwer.ipm.edu.mo/COMP122/20/

Bachelor of Science in Computing, School of Applied Sciences, Macao Polytechnic Institute

February 3, 2020

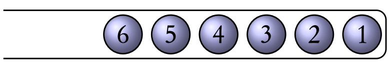
Outline

👁 *Textbook §6.1 – 6.2, 7.1.1 – 7.1.2.*

Stacks and LIFOs

- For a singly linked lists, if we keep inserting elements at the head position, and then keep removing the head elements. We have the following property:

  The last element inserted to the list is the first element deleted.
- This property is called "Last-In, First-Out", or LIFO.
- Another common name for this kind of structure is "stack".
- Stacks are abstract, any collections of elements that have the LIFO property can be regarded as stacks, or LIFOs.



The *Stack* ADT

Formally, a stack is an *abstract data type* (ADT) such that an instance supports the following operations.

- *push*(*self*, *x*) — add element *x* to the top of the stack.
- *pop*(*self*) — remove and return the top element from the stack, an error occurs if the stack is empty.
- *top*(*self*) — return a reference to the top element of the stack, without removing it, an error occurs if the stack is empty.
- __*bool*__(*self*) — return `True` if the stack contains some elements, `False` if empty.

The singly linked list *LnLs* supports all the listed operations, thus, can be used directly as a stack.

Array-Based Stacks

- We can also use an array-based list as the storage of a stack, eliminating the node creation at each "push" for a linked list.
- System stacks manipulated by processors are implemented as even a fixed length array to achieve high efficiency.
- Since an array-based list can only be extended at the end, we push new elements by appending.
- Accordingly, we pop an element from the stack by deleting and returning the last item of the list.

Array-Based Stacks — Code

```
class AStack:
    def __init__(self):
        self.a = []
    def __bool__(self):
        return self.a != []
    def top(self):
        return self.a[-1]
    def push(self, x):
        self.a.append(x)
    def pop(self):
        x = self.top()
        del self.a[-1]
        return x
```

Applications of Stacks

Direct applications:

- Page-visited history in a Web browser.
- Undo sequence in a text editor.
- Chain of method calls in the Java Virtual Machine.

Indirect applications:

- Auxiliary data structure for algorithms.
- Component of other data structures.

Parentheses Matching

- Each "(", "{", or "[" must be paired with a matching ")", "}", or "]".

```
  Correct : ( )(( )){([( )])}
  Correct : ((( )(( ))){([( )])})
Incorrect : )(( )){([( )])}
Incorrect : ({[ ])}
Incorrect : ([()]
```

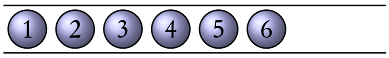
- We use a stack to store opening symbols. We scan the string, and for each character c,
  - if c is an opening symbol, we push it onto the stack.
  - if c is a closing symbol, we pop the stack and see if it matches c. If we cannot pop the stack or the symbol popped out does not match, we declare a mismatch.
- When the scanning completes without mismatches, and the stack is finally empty, then the string is correct.

Parentheses Matching — Code

```python
def paren_match(s):
    st = AStack()
    for c in s:
        if c in ('(', '[', '{'):
            st.push(c)
        elif c in (')', ']', '}'):
            if not st or st.pop()+c not in ('()', '[]', '{}'):
                return False
    return not st
```

Queues and FIFOs

- Queues are linear structures that insert elements at one end and delete elements at the other end.
- The first element inserted to a queue is deleted first.
- Insertion at the rear end is called “enqueue” or “push back”, deletion at the front end is called “dequeue” or “pop”.
- A queue is also called a FIFO (First-In, First-Out).
- Queues are also abstract, any collections of elements that have the FIFO property can be regarded as queues, or FIFOs.



The *Queue* ADT

Formally, a queue is an ADT such that an instance supports the following operations.

- *push_back*(*self*, *x*) — add element *x* to the end of the queue.
- *pop*(*self*) — remove and return the first element from the queue, an error occurs if the queue is empty.
- *top*(*self*) — return a reference to the first element of the queue, without removing it, an error occurs if the queue is empty.
- *__bool__*(*self*) — return `True` if the queue contains some elements, `False` if empty.

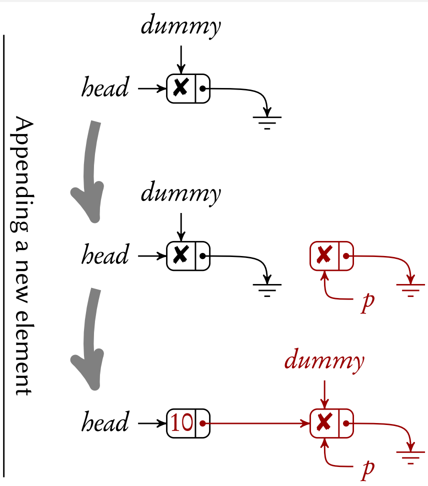
Applications of Queues

Direct applications:

- Waiting lists, bureaucracy.
- Access to shared resources (e.g., printer).
- Multiprogramming.

Indirect applications

- Auxiliary data structure for algorithms.
- Component of other data structures.



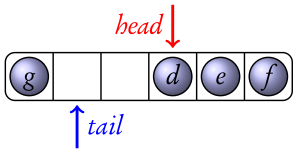
Linked List-Based Queues

- We need to append a node next to the last node in order to *push_back*.
- A reference to the last node must be recorded.
- However, when the linked list is empty, we do not have the last node.
- To unify the handling of the two cases, we introduce a *dummy* node as the last node.
- When we *push_back*, we simply put the new element into the old dummy node, and link it to a new dummy node.

```
class LQueue:
    def __init__(self):
        self.head = self.dummy = Node(None, None)
    def __bool__(self):
        return self.head is not self.dummy
    def push_back(self, x):
        self.dummy.elm, self.dummy.nxt = x, Node(None, None)
        self.dummy = self.dummy.nxt
```


Circular Buffer-Based Queues

- We can also implement a queue using an array-based list.
- Instead of shifting elements (which is expensive), we mark the position of the first element (*head*), and
- the position next to the last element, the first vacant cell (*tail*).
  - The positions increase one way! At a certain time, they must reach the end of the array.
- When *head* becomes greater, the smaller positions are available. So we can wrap both markers back to the beginning when they reach the end of the array. This is known as a *circular buffer*.



Full and Empty Circular Buffers

- How many elements can be held in a circular buffer of size n?
- The number of elements in a circular buffer can be computed using a modular subtraction:

$$(tail - head)\ \%\ n.$$

- The range of the above expression is from 0 to $n-1$. Therefore, there are at most $n-1$ elements that can be held without additional information.
- How to wrap?

$$head \leftarrow (head + 1)\ \%\ n.$$

- How to detect if a circular buffer is empty or full?
  - Obviously, if *head* equals *tail*, there is no element.
  - If $(tail + 1)\ \%\ n$ reaches *head*, the buffer is full.
  - When the queue is full, we can extend the underlying list at *tail*, and adjust *head* accordingly.

Circular Buffer-Based Queues — Code

```
class AQueue:
    def __init__(self):
        self.a = [None]*16
        self.n = len(self.a)
        self.head = self.tail = 0

    def __bool__(self):
        return self.head != self.tail

    def __len__(self):
        return (self.tail-self.head)%self.n

    def __iter__(self):
        for i in range(len(self)):
            yield self.a[(self.head+i)%self.n]
```


Circular Buffer-Based Queues — Code (2)

```
    def top(self):
        if not self:
            raise IndexError
        return self.a[self.head]

    def pop(self):
        x = self.top()
        self.a[self.head] = None
        self.head = (self.head+1)%self.n
        return x
```


Circular Buffer-Based Queues — Code (3)

```
    def push_back(self, x):
        l = len(self)
        if l+1 == self.n:   # back-end list is full
            self.a[self.tail:self.tail] = [None]*self.n
            self.n = len(self.a)
            self.head = (self.tail-l)%self.n

        self.a[self.tail] = x
        self.tail = (self.tail+1)%self.n
```