

Outline

1 Stacks

2 Queues

3 Symbol Tables

A stack is an iterable collection that is based on the last-in-first-out (LIFO) policy

A stack is an iterable collection that is based on the last-in-first-out (LIFO) policy

An iterable data type ${\tt ArrayStack}$ that represents a stack

■ ArrayStack			
ArrayStack()	initialize an empty stack s		
s.isEmpty()	is s empty?		
len(s)	number of elements in s		
s.push(item)	push item on top of s		
s.peek()	peek and return <i>item</i> on top of s		
s.pop()	pop and return the item on top of s		
iter(s)	an iterator over the elements of s		

Program: reverse.py

Program: reverse.py

• Standard input: a sequence of strings

Program: reverse.py

- Standard input: a sequence of strings
- Standard output: the strings in reverse order

Program: reverse.py

- Standard input: a sequence of strings
- Standard output: the strings in reverse order

>_ ~/workspace/ipp/program

```
$ python3 reverse.py
b o 1 t o n
<ctrl-d>
n o t 1 o b
```

```
from arraystack import ArrayStack
import stdio

def main():
    stack = ArrayStack()
    while not stdio.isEmpty():
        s = stdio.readString()
        stack.push(s)
    for s in stack:
        stdio.write(s + ' ')
    stdio.write(h()

if __name__ == '__main__':
    main()
```

```
arraystack.py
import stdio
class ArrayStack:
    def init (self):
        self._a = []
    def isEmpty(self):
        return len(self) == 0
    def __len__(self):
        return len(self. a)
    def push(self. item):
        self._a.append(item)
    def peek(self):
        if self.isEmptv():
            raise Exception('Stack underflow')
        return self. a[-1]
    def pop(self):
        if self.isEmpty():
            raise Exception('Stack underflow')
        return self._a.pop(-1)
    def __iter__(self):
        return iter(reversed(self._a))
def _main():
    stack = ArravStack()
    while not stdio.isEmpty():
        item = stdio.readString()
        if item != '-':
            stack.push(item)
        elif not stack.isEmptv():
```

```
🗷 arraystack.py
            stdio.write(str(stack.pop()) + ' ')
    stdio.writeln('(' + str(len(stack)) + ' left on stack)')
if __name__ == '__main__':
    _main()
```



A queue is an iterable collection that is based on the first-in-first-out (FIFO) policy

A queue is an iterable collection that is based on the first-in-first-out (FIFO) policy

An iterable data type ${\mbox{\sc arrayQueue}}$ that represents a queue

I ArrayQueue			
ArrayQueue()	initialize an empty queue q		
q.isEmpty()	is q empty		
len(q)	number of elements in q		
q.enqueue(item)	add $item$ to the end of q		
q.peek()	peek and return the first item of q		
q.dequeue()	remove and return the first item of q		
iter(q)	an iterator over the elements of q		



 $Program: {\tt kthfromlast.py}$

Program: kthfromlast.py

• Command-line input: k (int)

Program: kthfromlast.py

ullet Command-line input: k (int)

• Standard input: sequence of strings

 $Program: {\tt kthfromlast.py}$

• Command-line input: *k* (int)

• Standard input: sequence of strings

• Standard output: kth string from the end

 $Program: \ {\tt kthfromlast.py}$

• Command-line input: k (int)

• Standard input: sequence of strings

ullet Standard output: kth string from the end

>_ ~/workspace/ipp/programs

```
$ python3 kthfromlast.py 5
she sells sea shells on the sea shore
<ctrl-d>
shells
```



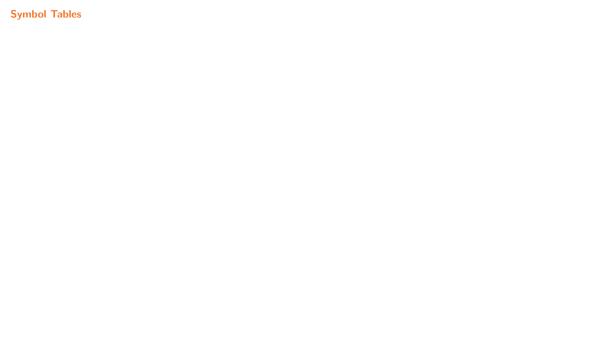
```
☑ kthfromlast.py
from arrayqueue import ArrayQueue
import stdio
import sys
def main():
    k = int(sys.argv[1])
    queue = ArravQueue()
    while not stdio.isEmpty():
        s = stdio.readString()
        queue.enqueue(s)
    n = len(queue)
    for i in range(1, n - k + 1):
        queue.dequeue()
    stdio.writeln(queue.peek())
if __name__ == '__main__':
    main()
```



```
☑ ArrayQueue.py
import stdio
class ArrayQueue:
    def init (self):
        self._a = []
    def isEmpty(self):
        return len(self) == 0
    def __len__(self):
        return len(self. a)
    def enqueue(self. item):
        self._a.append(item)
    def peek(self):
        if self.isEmptv():
            raise Exception('Queue underflow')
        return self. a[0]
    def dequeue(self):
        if self.isEmpty():
            raise Exception('Queue underflow')
        return self._a.pop(0)
    def __iter__(self):
        return iter(self._a)
def _main():
    queue = ArravQueue()
    while not stdio.isEmpty():
        item = stdio.readString()
        if item != '-':
            queue.enqueue(item)
        elif not queue.isEmptv():
```

```
☑ ArrayQueue.py

            stdio.write(str(queue.dequeue()) + ' ')
    stdio.writeln('(' + str(len(queue)) + ' left on queue)')
if __name__ == '__main__':
    _main()
```



A symbol table is a data structure for key-value pairs that supports two operations: insert (put) a new pair into the table and search (get) the value associated with a given key

A symbol table is a data structure for key-value pairs that supports two operations: insert (put) a new pair into the table and search (get) the value associated with a given key

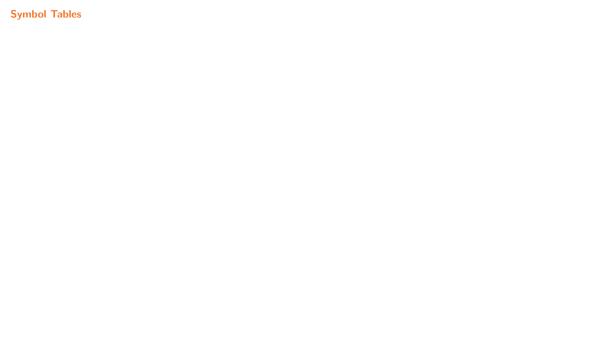
No duplicate keys are allowed; when we put a key-value pair into a table already containing that key (and an associated value), the new value replaces the old one

A symbol table is a data structure for key-value pairs that supports two operations: insert (put) a new pair into the table and search (get) the value associated with a given key

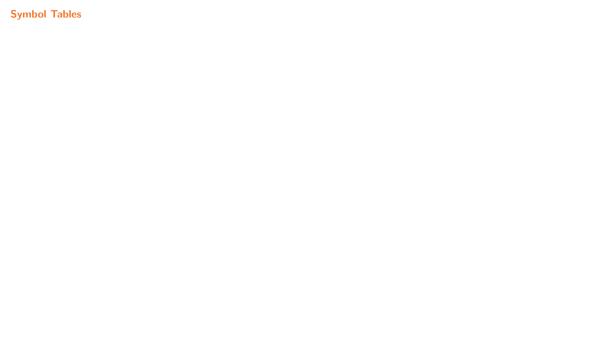
No duplicate keys are allowed; when we put a key-value pair into a table already containing that key (and an associated value), the new value replaces the old one

Applications

Application	Purpose	Key	Value
dictionary	find definition	word	definition
book index	find relevant pages	term	list of page numbers
web search	find relevant web pages	keyword	list of page names



```
■ SymbolTable
                constructs an empty symbol table s
 SymbolTable()
                returns True if s is empty, and False otherwise
 s.isEmpty()
                returns the number of key-value pairs in s
 len(s)
                returns True if s contains key, and False otherwise
 kev in s
                returns the value associated with key in s
 s[key]
 s[kev] = val
                inserts the pair key/val into s
                returns the keys in s as an iterable object
 s.keys()
                returns the values in s as an iterable object
 s.values()
```



 $Program: \ {\tt frequency counter.py}$

Program: frequencycounter.py

• Command-line input: minLen (int)

Program: frequencycounter.py

• Command-line input: minLen (int)

• Standard input: a sequence of words

Program: frequencycounter.py

- Command-line input: minLen (int)
- Standard input: a sequence of words
- Standard output: for the words that are at least as long as *minLen*, writes the total word count, the number of distinct words, and the most frequent word

Program: frequencycounter.py

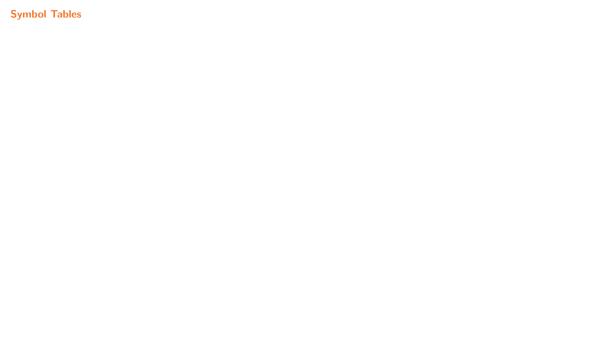
• Command-line input: minLen (int)

• Standard input: a sequence of words

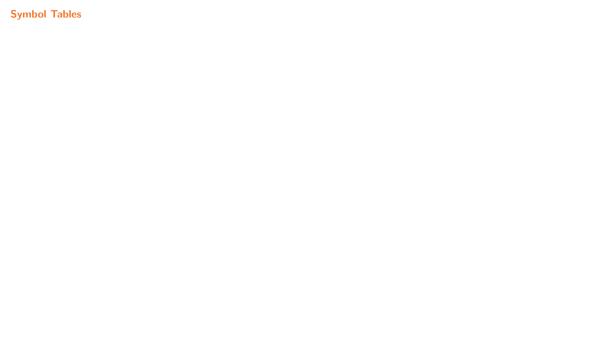
• Standard output: for the words that are at least as long as *minLen*, writes the total word count, the number of distinct words, and the most frequent word

>_ ~/workspace/ipp/programs

```
$ python3 frequencycounter.py 8 < ../data/tale.txt
Word count: 13525
Distinct word count: 4371
Most frequent word: business (134 repetitions)</pre>
```



```
frequencycounter.py
from symboltable import SymbolTable
import stdio
import sys
def main().
    minLen = int(sys.argv[1])
    distinct, words = 0.0
    st = SymbolTable()
    while not stdio.isEmptv():
        word = stdio.readString()
        if len(word) < minLen:
            continue
        words += 1
        if word in st:
            st[word] += 1
        else:
            st[word] = 1
            distinct += 1
    maxFreq = 0
    maxFreqWord = ''
    for word in st.kevs():
        if st[word] > maxFreq:
            maxFreq = st[word]
            maxFreqWord = word
    stdio_writeln('Word count: ' + str(words))
    stdio.writeln('Distinct word count: ' + str(distinct))
    stdio.writef('Most frequent word: %s (%d repetitions)\n', maxFreqWord, maxFreq)
if name == ' main ':
    main()
```



A dictionary (an object of the built-in mapping type $_{\tt dlct}$) is an unordered set of key-value pairs, with the requirement that the keys be unique

A dictionary (an object of the built-in mapping type dict) is an unordered set of key-value pairs, with the requirement that the keys be unique

The simplest way to create a dictionary is to place comma-separated key-value pairs (the key and value within a pair are separated by a colon) between matching curly brackets

A dictionary (an object of the built-in mapping type $_{\tt dlct}$) is an unordered set of key-value pairs, with the requirement that the keys be unique

The simplest way to create a dictionary is to place comma-separated key-value pairs (the key and value within a pair are separated by a colon) between matching curly brackets

Example (days of the week)

```
dow = {0 : 'Sun', 1 : 'Mon', 2 : 'Tue', 3 : 'Wed', 4 : 'Thu', 5 : 'Fri', 6 : 'Sat'}
```

A dictionary (an object of the built-in mapping type dict) is an unordered set of key-value pairs, with the requirement that the keys be unique

The simplest way to create a dictionary is to place comma-separated key-value pairs (the key and value within a pair are separated by a colon) between matching curly brackets

Example (days of the week)

```
dow = {0 : 'Sun', 1 : 'Mon', 2 : 'Tue', 3 : 'Wed', 4 : 'Thu', 5 : 'Fri', 6 : 'Sat'}
```

The len() function can be used to obtain the number of key-value pairs in a dictionary; in the above example, len(dow) returns 7

A dictionary (an object of the built-in mapping type $_{ exttt{dict}}$) is an unordered set of key-value pairs, with the requirement that the keys be unique

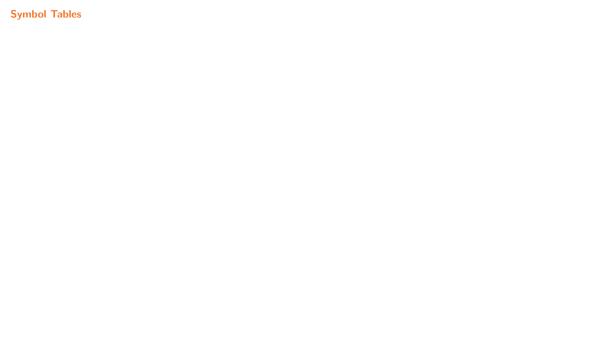
The simplest way to create a dictionary is to place comma-separated key-value pairs (the key and value within a pair are separated by a colon) between matching curly brackets

Example (days of the week)

```
dow = {0 : 'Sun', 1 : 'Mon', 2 : 'Tue', 3 : 'Wed', 4 : 'Thu', 5 : 'Fri', 6 : 'Sat'}
```

The len() function can be used to obtain the number of key-value pairs in a dictionary; in the above example, len(dow) returns 7

The comparison operator in can be used to check if a particular key exists in a dictionary; in the above example, 5 in dow evaluates to True, whereas 42 in dow evaluates to False



If d is a dictionary, then d[key] returns the value associated with key, and raises KeyError() if the key doesn't exist in the dictionary; in the above example, dow[5] returns 'Fri', whereas dow[42] raises KeyError()

If a is a dictionary, then d[key] returns the value associated with key, and raises KeyError() if the key doesn't exist in the dictionary; in the above example, dow[5] returns 'Fri', whereas dow[42] raises KeyError()

The following statement inserts the key-value pair $_{\text{key}}/_{\text{val}}$ into a dictionary $_{\text{d}}$

```
d[key] = val
```

Note that if key is already in d, then its value is updated to val

If a is a dictionary, then d[key] returns the value associated with key, and raises KeyError() if the key doesn't exist in the dictionary; in the above example, dow[5] returns 'Fri', whereas dow[42] raises KeyError()

The following statement inserts the key-value pair $_{\text{key}}/_{\text{val}}$ into a dictionary $_{\text{d}}$

```
d[key] = val
```

Note that if $_{\tt key}$ is already in $_{\tt d},$ then its value is updated to $_{\tt val}$

Example (add/update dow)

```
dow[7] = 'Error'
dow[5] = 'Friday'
```

If a is a dictionary, then d[key] returns the value associated with key, and raises KeyError() if the key doesn't exist in the dictionary; in the above example, dow[5] returns 'Fri', whereas dow[42] raises KeyError()

The following statement inserts the key-value pair $_{\text{key}}/_{\text{val}}$ into a dictionary $_{\text{d}}$

```
d[key] = val
```

Note that if $_{\hbox{\scriptsize key}}$ is already in $_{\hbox{\scriptsize d}},$ then its value is updated to $_{\hbox{\scriptsize val}}$

Example (add/update dow)

```
dow(7) = 'Error'
dow(5) = 'Friday'
```

If a is a dictionary, then d.keys() and d.values() respectively return the keys and values in a as an interable object

If a is a dictionary, then d[key] returns the value associated with key, and raises KeyError() if the key doesn't exist in the dictionary; in the above example, dow[6] returns 'Fri', whereas dow[42] raises KeyError()

The following statement inserts the key-value pair $_{\text{key}}/_{\text{val}}$ into a dictionary $_{\text{d}}$

```
d[key] = val
```

Note that if $_{\tt key}$ is already in $_{\tt d},$ then its value is updated to $_{\tt val}$

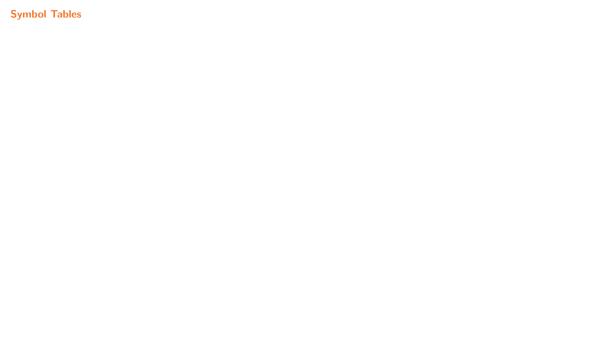
Example (add/update dow)

```
dow[7] = 'Error'
dow[8] = 'Friday'
```

If a is a dictionary, then d.keys() and d.values() respectively return the keys and values in a as an interable object

Example (iterate over keys and values of ${\tiny \texttt{dow}})$

```
for key in dow.keys():
    stdio.writeln(key + '->' + dow[key]
for val in dow.values():
    stdio.writeln(val)
```



```
g symboltable.py
import stdio
class SymbolTable:
    def init (self):
        self._st = {}
    def isEmpty(self):
        return len(self. st) == 0
    def __len__(self):
        return len(self. st)
    def __contains__(self, key):
        return key in self._st
    def __getitem__(self, key):
        return self. st[kev]
    def setitem (self. kev. val):
        self._st[key] = val
    def kevs(self):
        return iter(self._st.kevs())
    def values(self):
        return iter(self._st.values())
def _main():
    st = SymbolTable()
    st['Gautama'] = 'Siddhartha'
    st['Darwin'] = 'Charles'
    st['Einstein'] = 'Albert'
    stdio.writeln(st['Gautama'])
    stdio_uriteln(st['Darwin'])
    stdio.writeln(st['Einstein'])
```

```
if 'Einstein' in st:
       stdio.writeln('Einstein found')
   else:
       stdio.writeln('Einstein not found')
   if 'Newton' in st:
       stdio.writeln('Newton found')
   else:
       stdio.writeln('Newton not found')
   for key in st.keys():
       stdio.writeln(key + ': ' + st[key])
   for value in st.values():
       stdio.writeln(value)
if __name__ == '__main__':
   _main()
```