Algorithms and Data Structures for Data Science
Stacks and Queues (and Sets)

CS 277
Brad Solomon

February 13, 2023
Learning Objectives

- Explore tradeoffs in data structures
- Introduce the stack and queue
- Introduce sets
Tradeoffs in data structures

As we progress in the class, we will see that $O(n)$ isn’t very good.

Take searching for a specific list value:
Tradeoffs in data structures

Getting the size of a linked list has a Big O of:
Genome assembly databases are growing rapidly. The sequence content in each new assembly can be largely redundant with previous ones, but this is neither conceptually nor algorithmically easy to measure. We propose new methods and a new tool called DandD that addresses the question of how much new sequence is gained when a sequence collection grows. DandD can describe how much human structural variation is being discovered in each new human genome assembly and when discoveries will level off in the future. DandD uses a measure called $\delta$ ("delta"), developed initially for data compression. Computing $\delta$ directly requires counting k-mers, but DandD can rapidly estimate it using genomic sketches. We also propose $\delta$ as an alternative to k-mer-specific cardinalities when computing the Jaccard coefficient, avoiding the pitfalls of a poor choice of k. We demonstrate the utility of DandD’s functions for estimating $\delta$, characterizing the rate of pangenome growth, and computing allpairs similarities using k-independent Jaccard. DandD is open source software available at: https://github.com/jessicabonnie/dandd.

Jessica Bonnie et al. DandD: efficient measurement of sequence growth and similarity
Tradeoffs in data structures

I want a list that can add and remove in $O(1)$

I am willing to make random access impossible to do so
Stack Data Structure

A stack stores an ordered collection of objects (like a list)

However you can only do two operations:

**Push**: Put an item on top of the stack

**Pop**: Remove the top item of the stack (and return it)

```
push(3);  push(5);  pop();  push(2)
```
Stack Data Structure

The stack is a **last in — first out** data structure (LIFO)

```python
def reverse(inList):
    s = stack()
    for v in inList:
        s.push(v)
    out = []
    while not s.empty():
        out.append(s.pop())
    return out
```

reverse([3, 4, 5, 6, 7, 8])
Stack Data Structure

The Python list has all the necessary stack operations!

```
1 def push(self, val):
    self.push_count += 1
    self.data.append(val)
2
3 def pop(self):
    self.pop_count += 1
    if self.__len__() > 0:
        return self.data.pop()
4
5
6
7
8
9

Stack s
s.push(3)
s.push(8)
s.push(4)
s.pop()
s.push(7)
s.pop()
s.pop()
s.push(2)
s.push(1)
s.push(3)
s.push(5)
s.pop()
s.push(9)
```
Stack Data Structure

The stack is also easily implemented as a linked list

```
push(X)

head
```

```
C -> S -> 2 -> 7 -> 7 -> None
```
Stack Data Structure

The stack is also easily implemented as a linked list.

```
head
C → S → 2 → 7 → 7 → None
```
def Happy():
    return "Happy"

def Little():
    return "Little"

def Trees():
    return "Trees"

def LittleTrees():
    return Little() + Trees()

def BobRoss():
    return Happy() + LittleTrees()

if __name__ == '__main__':
    print(BobRoss())
Queue Data Structure

A **queue** stores an ordered collection of objects (like a list)

However you can only do two operations:

**Enqueue**: Put an item at the back of the queue

**Dequeue**: Remove and return the front item of the queue

enqueue(3); enqueue(5); dequeue(); enqueue(2)
Queue Data Structure

The queue is a **first in — first out** data structure (FIFO)

What data structure excels at removing from the front?

Can we make that same data structure good at inserting at the end?
Queue Data Structure

The queue is a **first in — first out** data structure (FIFO)
Queue Data Structure

The queue is a **first in — first out** data structure (FIFO)
Queue Data Structure

An array can implement a queue as well!

We just need to track three numbers:

Front

Capacity

Size
**Queue Data Structure**

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>

Queue q
- q.enqueue(3)
- q.enqueue(8)
- q.enqueue(4)
- q.dequeue()
- q.enqueue(7)
- q.dequeue()
- q.dequeue()
- q.enqueue(2)
- q.enqueue(1)
- q.enqueue(3)
- q.enqueue(5)
- q.dequeue()
- q.enqueue(9)

**Front:**

**Capacity:**

**Size:**
Queue Data Structure

The array implementation treats the allocated memory as a circle.
Queue Array Resizing

Queue q
...
# Fill queue
...
q.enqueue(8)

Front: 2
Capacity: 8
Size: 8
Stacks and Queues

- Enqueue
- Dequeue

- Push
- Pop
<table>
<thead>
<tr>
<th>Activity</th>
<th>Time x1 billion</th>
<th>Like</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache reference</td>
<td>0.5 seconds</td>
<td>Heartbeat</td>
</tr>
<tr>
<td>Branch mispredict</td>
<td>5 seconds</td>
<td>Yawn</td>
</tr>
<tr>
<td>L2 cache reference</td>
<td>7 seconds</td>
<td>Long yawn</td>
</tr>
<tr>
<td>Mutex lock/unlock</td>
<td>25 seconds</td>
<td>Make coffee</td>
</tr>
<tr>
<td>Main memory reference</td>
<td>100 seconds</td>
<td>Brush teeth</td>
</tr>
<tr>
<td>Compress 1K bytes</td>
<td>50 minutes</td>
<td>TV show</td>
</tr>
<tr>
<td>Send 2K bytes over 1 Gbps network</td>
<td>5.5 hours</td>
<td>(Brief) Night's sleep</td>
</tr>
<tr>
<td>SSD random read</td>
<td>1.7 days</td>
<td>Weekend</td>
</tr>
<tr>
<td>Read 1 MB sequentially from memory</td>
<td>2.9 days</td>
<td>Long weekend</td>
</tr>
<tr>
<td>Read 1 MB sequentially from SSD</td>
<td>11.6 days</td>
<td>2 weeks for delivery</td>
</tr>
<tr>
<td>Disk seek</td>
<td>16.5 weeks</td>
<td>Semester</td>
</tr>
<tr>
<td>Read 1 MB sequentially from disk</td>
<td>7.8 months</td>
<td>Human gestation</td>
</tr>
<tr>
<td>Above two together</td>
<td>1 year</td>
<td></td>
</tr>
<tr>
<td>Send packet CA-&gt;Netherlands-&gt;CA</td>
<td>4.8 years</td>
<td>Ph.D.</td>
</tr>
</tbody>
</table>

(Care of https://gist.github.com/hellerbarde/2843375)
class queue:
    def __init__(self):
        self.data = []

    def enqueue(self, val):
        self.data.append(val)

    def dequeue(self):
        if len(self.data) > 0:
            return self.data.pop(0)

def enqueue(self, val):
    i = (self.front + self.size) % self.capacity
    self.data[i] = val
    self.size += 1

    if self.size == self.capacity:
        temp = [None] * self.capacity * 2
        for i in range(self.size):
            pos = (self.front + i) % self.capacity
            temp[i] = self.data[pos]

        self.front = 0
        self.data = temp
        self.capacity = len(temp)
Tradeoffs in data structures

I want a data structure that can add, remove, and find items in $O(1)$ *

I am willing to remove the ‘ordered’ property of my collection to do this

I am willing to remove the ability to store duplicate elements to do this
Set “Data Structure”

A set stores an unordered collection of objects with no duplicates

Genome assembly databases are growing rapidly. The sequence content in each new assembly can be largely redundant with previous ones, but this is neither conceptually nor algorithmically easy to measure. We propose new methods and a new tool called DandD that addresses the question of how much new sequence is gained when a sequence collection grows. DandD can describe how much human structural variation is being discovered in each new human genome assembly and when discoveries will level off in the future. DandD uses a measure called $\delta$ (“delta”), developed initially for data compression. Computing $\delta$ directly requires counting k-mers, but DandD can rapidly estimate it using genomic sketches. We also propose $\delta$ as an alternative to k-mer-specific cardinalities when computing the Jaccard coefficient, avoiding the pitfalls of a poor choice of k. We demonstrate the utility of DandD’s functions for estimating $\delta$, characterizing the rate of pangenome growth, and computing allpairs similarities using k-independent Jaccard. DandD is open source software available at: https://github.com/jessicabonnie/dandd.
Sets in Python: Constructor

The set constructor `set()` takes a list or tuple as input.

```python
s1 = set([1,2,3,4])
s2 = set((3,4,5,6))
```
Sets in Python: Add

\textbf{Add}(x) \textit{adds object} x \textit{to the set; it does nothing if} x \textit{is already present}

```
mySet = set()
mySet.add(1)
mySet.add(1)
mySet.add(3)
```
Sets in Python: Remove

**Remove**(x) removes the object x from the set if it is present.

If x does not exist, it will crash.

```python
mySet = set([1,2,3,4,5])
mySet.remove(3)
print(mySet)
mySet.remove(10)
```
Sets in Python: Data Access

Sets have no indices (no order of objects). We can only access by:

1. Looping through a set for each element
2. Looking up a specific element in our set

```python
mySet = set([1,2,3,4,5])
for obj in mySet:
    print(obj)
print(10 in mySet)
```
Sets in Python: Implementation

A Python set is implemented using a hash table.

**Claim:** A hash table has direct lookup, add, and remove in $O(1)$ *
Set Operations

\[ A = \{1, 2, 3, 4\} \quad B = \{3, 4, 5, 6, 7\} \]

- Union: \( A \cup B \)
- Intersection: \( A \cap B \)
- Difference: \( A / B \)
- Symmetric difference: \( A \triangle B \)
Set Similarity

How can we describe how similar two sets are?
Set Similarity

How can we describe how similar two sets are?
Set Similarity

To measure similarity of $A$ & $B$, we need both a measure of how similar the sets are but also the total size of both sets.

$$J \equiv \frac{|A \cap B|}{|A \cup B|}$$

$J$ is the Jaccard coefficient.
Set Similarity

\[
\frac{|A \cap B|}{|A \cup B|} = 0
\]

\[
\frac{|A \cap B|}{|A \cup B|} = 1
\]

\[
0 < \frac{|A \cap B|}{|A \cup B|} < 1
\]
Set Similarity

\[ A = \{1, 2, 3, 4\} \quad B = \{3, 4, 5, 6, 7\} \]

\[ J = \frac{|A \cap B|}{|A \cup B|} = \]
Set Similarity

\[ A = \{1, 2, 3, 4\} \quad B = \{3, 4, 5, 6, 7\} \]

\[ J = \frac{|A \cap B|}{|A \cup B|} = \frac{|\{3, 4\}|}{|\{1, 2, 3, 4, 5, 6, 7\}|} = \frac{2}{7} \]