Lecture 09, 12 September 2023

Arrays ¶

- Contiguous block of memory
- Typically size is declared in advance, all values are uniform
- `a[0]` points to first memory location in the allocated block
- Locate `a[i]` in memory using index arithmetic
  - Skip `i` blocks of memory, each block's size determined by value stored in array
- **Random access** -- accessing the value at `a[i]` does not depend on `i`
- Useful for procedures like sorting, where we need to swap out of order values `a[i]` and `a[j]`
  - `a[i], a[j] = a[j], a[i]`
  - Cost of such a swap is constant, independent of where the elements to be swapped are in the array
- Inserting or deleting a value is expensive
- Need to shift elements right or left, respectively, depending on the location of the modification

Lists

- Each location is a *cell*, consisiting of a value and a link to the next cell
  - Think of a list as a train, made up of a linked sequence of cells
- The name of the list `l` gives us access to `l[0]`, the first cell
- To reach cell `l[i]`, we must traverse the links from `l[0]` to `l[1]` to `l[2]` ... to `l[i-1]` to `l[i]`
  - Takes time proportional to `i`
- Cost of swapping `l[i]` and `l[j]` varies, depending on values `i` and `j`
- On the other hand, if we are already at `l[i]` modifying the list is easy
  - *Insert* - create a new cell and reroute the links
  - *Delete* - bypass the deleted cell by rerouting the links
- Each insert/delete requires a fixed amount of local "plumbing", independent of where in the list it is performed

Dictionaries

- Values are stored in a fixed block of size m
- Keys are mapped to $\{0, 1, \ldots, m-1\}$
- Hash function $h : K \rightarrow S$ maps a *large* set of keys K to a *small* range S
- Simple hash function: interpret $k \in K$ as a bit sequence representing a number n_k in binary, and compute $n_k \bmod m$, where $|S| = m$
- Mismatch in sizes means that there will be *collisions* -- $k_1 \neq k_2$, but $h(k_1) = h(k_2)$
- A good hash function maps keys "randomly" to minimize collisions
- Hash can be used as a *signature* of authenticity
  - Modifying k slightly will drastically alter $h(k)$
  - No easy way to reverse engineer a k' to map to a given $h(k)$
  - Use to check that large files have not been tampered with in transit, either due to network errors or malicious intervention
- Dictionary uses a hash function to map key values to storage locations
- Lookup requires computing $h(k)$ which takes roughly the same time for any k
  - Compare with computing the offset `a[i]` for any index `i` in an array
- Collisions are inevitable, different mechanisms to manage this, which we will not discuss now
- Effectively, a dictionary combines flexibility with random access

Lists in Python

- Flexible size, allow inserting/deleting elements in between
- However, implementation is an array, rather than a list
- Initially allocate a block of storage to the list
- When storage runs out, double the allocation
- `l.append(x)` is efficient, moves the right end of the list one position forward within the array
- `l.insert(0,x)` inserts a value at the start, expensive because it requires shifting all the elements by 1
- We will run experiments to validate these claims

Measuring execution time

- Call `time.perf_counter()`
- Actual return value is meaningless, but difference between two calls measures time in seconds

In [1]:
```python
import time
```

- 10^7 appends to an empty Python list

In [2]:
```python
start = time.perf_counter()
l = []
for i in range(10000000):
    l.append(i)
elapsed = time.perf_counter() - start
print(elapsed)
```

```
3.1834037989901844
```

- Doubling the work approximately doubles the time, linear

In [3]:
```
start = time.perf_counter()
l = []
for i in range(20000000):
    l.append(i)
elapsed = time.perf_counter() - start
print(elapsed)
```

```
5.753009960986674
```

- 10^5 inserts at the beginning of a Python list

In [4]:
```
start = time.perf_counter()
l = []
for i in range(100000):
    l.insert(0,i)
elapsed = time.perf_counter() - start
print(elapsed)
```

```
5.5166299150150735
```

- Doubling and tripling the work multiplies the time by 4 and 9, respectively, so quadratic

In [5]:
```
start = time.perf_counter()
l = []
for i in range(200000):
    l.insert(0,i)
elapsed = time.perf_counter() - start
print(elapsed)
```

```
17.979196411994053
```

In [6]:
```
start = time.perf_counter()
l = []
for i in range(300000):
    l.insert(0,i)
elapsed = time.perf_counter() - start
print(elapsed)
```

```
43.46195148699917
```

- Creating 10^7 entries in an empty dictionary

In [7]:
```
start = time.perf_counter()
d = {}
for i in range(10000000,0,-1):
    d[i] = i
elapsed = time.perf_counter() - start
print(elapsed)
```

```
3.8069355089974124
```

- Doubling the operations, doubles the time, so linear
- Dictionaries are effectively random access

In [8]:
```
start = time.perf_counter()
d = {}
for i in range(20000000,0,-1):
    d[i] = i
elapsed = time.perf_counter() - start
print(elapsed)
```

```
9.057193082000595
```

Implementing a "real" list using dictionaries

In [9]:
```python
def createlist():  # Equivalent of l = [] is l = createlist()
  return({})

def listappend(l,x):
  if l == {}:
    l["value"] = x
    l["next"] = {}
    return

  node = l
  while node["next"] != {}:
    node = node["next"]

  node["next"]["value"] = x
  node["next"]["next"] = {}
  return

def listinsert(l,x):
  if l == {}:
    l["value"] = x
    l["next"] = {}
    return

  newnode = {}
  newnode["value"] = l["value"]
  newnode["next"] = l["next"]
  l["value"] = x
  l["next"] = newnode
  return


def printlist(l):
  print("{",end="")

  if l == {}:
    print("}")
    return
  node = l

  print(node["value"],end="")
  while node["next"] != {}:
    node = node["next"]
    print(",",node["value"],end="")
  print("}")
  return
```

- Display a small list as nested dictionaries

In [10]:
```python
start = time.perf_counter()
l = createlist()
for i in range(10):
    listappend(l,i)
elapsed = time.perf_counter() - start
print(elapsed)
print(l)
```

```
0.020103318995097652
{'value': 0, 'next': {'value': 1, 'next': {'value': 2, 'next': {'value': 3, 'next': {'value': 4, 'next': {'value': 5, 'nex
t': {'value': 6, 'next': {'value': 7, 'next': {'value': 8, 'next': {'value': 9, 'next': {}}}}}}}}}}}
```

- Insert 10^7 elements at the beginning in this implementation of a list

In [11]:
```python
start = time.perf_counter()
l = createlist()
for i in range(1000000):
    listinsert(l,i)
elapsed = time.perf_counter() - start
print(elapsed)
```

```
3.375442454998847
```

- Doubling the work doubles the time, so linear

In [12]:
```python
start = time.perf_counter()
l = createlist()
for i in range(2000000):
    listinsert(l,i)
elapsed = time.perf_counter() - start
print(elapsed)
```

```
6.131248404999496
```

- Append 10^4 elements in this implementation of a list

In [13]:
```python
start = time.perf_counter()
l = createlist()
for i in range(10000):
    listappend(l,i)
elapsed = time.perf_counter() - start
print(elapsed)
```

```
9.82448883599136
```

- Halving the work takes 1/4 of the time, so quadratic

In [14]:
```python
start = time.perf_counter()
l = createlist()
for i in range(5000):
    listappend(l,i)
elapsed = time.perf_counter() - start
print(elapsed)
```

```
2.685035665985197
```

Defining our own data structures

- We have implemented a "linked" list using dictionaries
- The fundamental functions like `listappend` , `listinsert` , `listdelete` modify the underlying list
- Instead of `mylist = {}` , we wrote `mylist = createlist()`
- To check empty list, use a function `isempty()` rather than `mylist == {}`
- Can we clearly separate the **interface** from the **implementation**
- Define the data structure in a more "modular" way

Set comprehension

- Defining new sets from old
- $\{x^2 \mid x \in \mathbb{Z}, x \geq 0 \wedge (x \bmod 2) = 0\}$
  - $x \in \mathbb{Z}$, generating set
  - $x \geq 0 \wedge (x \bmod 2) = 0$, filtering condition
  - x^2, output transformation
- More generally $\{f(x) \mid x \in S, p(x)\}$
  - generating set S
  - filtering predicate $p()$
  - transformer function $f()$

Can do this manually for lists

- List of squares of even numbers from 0 to 19
- Initialize output list as `[]`
- Run through a loop and append elements to output list

In [15]:
```python
evensqlist = []
for i in range(20):
  if i % 2 == 0:
    evensqlist.append(i*i)
print(evensqlist)
```

```
[0, 4, 16, 36, 64, 100, 144, 196, 256, 324]
```

Operating on each element of a list

- `map(f,l)` applies a function `f` to each element of a list `l`
- `filter(p,l)` extracts elements `x` from `l` for which `p(x)` is `True

In [16]:
```python
def even(x):
  return(x%2 == 0)

def odd(x):
  return(not(even(x)))

def square(x):
  return(x*x)

N = 20
l1 = list(range(N))
l2 = list(filter(odd,l1))  # Note that we can pass a function name as an argument
l3 = list(map(square,l1))

# Combine map and filter
l4 = list(map(square,filter(even,l1)))
```

In [17]:
```python
l1
```

Out[17]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

In [18]:
```
l2
```
Out[18]:
```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```
In [19]:
```
l3
```
Out[19]:
```
[0,
 1,
 4,
 9,
 16,
 25,
 36,
 49,
 64,
 81,
 100,
 121,
 144,
 169,
 196,
 225,
 256,
 289,
 324,
 361]
```
In [20]:
```
l4
```
Out[20]:
```
[0, 4, 16, 36, 64, 100, 144, 196, 256, 324]
```
List comprehension

- `[ f(x) for x in ... if p(x) ]`

In [21]:
```
[ square(x) for x in range(20) if even(x) ]
```
Out[21]:
```
[0, 4, 16, 36, 64, 100, 144, 196, 256, 324]
```
In [22]:
```
# A zero vector of length N
[ 0 for i in range(20)]    # The map function can be a constant function
```
Out[22]:
```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```
- List comprehension can be nested
- A 2 dimensional list : A list of M lists of N zeros

In [23]:
```
M,N = 3,5
onedim = [ 0 for i in range(N)]     #  A list of N zeros
twodim = [ [0 for i in range(N)] for j in range(M)]
```
In [24]:
```
onedim, twodim
```
Out[24]:
```
([0, 0, 0, 0, 0], [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]])
```
All Pythagorean triples with value less than n

- (x, y, z) such that $x^2 + y^2 = z^2, x, y, z \leq n$

Using nested loops

- Run through all possible `(x,y,z)`
- To avoid duplicates like `(3,4,5)` and `(4,3,5)` enumerate `y` starting from `x`
- `z` must be at least `y`, enumerate `z` starting from `y`

In [25]:
```
N = 20
triples = []
for x in range(1,N+1):
    for y in range(x,N+1):
        for z in range(y,N+1):
            if x*x + y*y == z*z:
                triples.append((x,y,z))
```
In [26]:
```
triples
```
Out[26]:
```
[(3, 4, 5), (5, 12, 13), (6, 8, 10), (8, 15, 17), (9, 12, 15), (12, 16, 20)]
```
Pythagorean triples via list comprehension

- Multiple generators for `x`, `y` and `z`
- As before start generator for `y` at `x` and generator for `z` at `y`

In [27]:
```
N = 20
[ (x,y,z) for x in range(1,N+1) for y in range(x,N+1) for z in range(y,N+1) if x*x + y*y == z*z]
```

Out[27]:
```
[(3, 4, 5), (5, 12, 13), (6, 8, 10), (8, 15, 17), (9, 12, 15), (12, 16, 20)]
```

Uses of list comprehension

List comprehension notation is compact and useful in a number of contexts

- Pull out all dictionary values where the keys satisfy some property: e.g. all marks below 50
  - `[ d[k] for k in d.keys() if p(k) ]`
- Symmetrically, keys whose values satisfy some property: e.g. all roll numbers where marks are below 50
  - `[ k for k in d.keys() if p(d[k]) ]`
- Or, extract (key,value) pairs of interest
  - `[ (k,d[k]) for k in d.keys() if p(d[k]) ]`