Algorithms for Data Science (Part 1 - Data Structures)

Daniel Lawson — University of Bristol

Lecture 08.2.1 (v1.0.1)

Signposting

- This lecture 8.2 of Algorithms for Data Science follows 8.1 on Analysing Algorithms
  - It is about some key algorithms that make Data Science approachable, even without a Big Data Platform.
  - These ideas are building blocks for statistical and machine-learning approaches for inference.
- The lecture is in two parts:
  - Part 1 Data Structures
  - Part 2 Algorithms
- This is Part 1, covering **Dynamic Data Structures**:
  - Hashing
  - Queues/Stacks
  - Linked Lists
  - Binary Trees/Heaps
  - Hash tables

ILOs

- ILO2 Be able to **use and apply basic machine learning** tools
- ILO4 Be able to use high throughput computing infrastructure and understand appropriate algorithms
- ILO5 Be able to reason about and conceptually align problems involving real data to appropriate theoretical methods and available methodology to correctly make inferences and decisions

Hash functions

- One of the most important components in good algorithmic design is the **hash**.
- Simply, a hash h is a map for $h(x) = u$ with:

$$x \in \mathcal{X} \to u \in \mathcal{U}[0, r).$$

- i.e., we map each item in the space $\mathcal{X}$ into the Uniform distribution on the integers $0, \ldots, r-1$.
- Each item will always map to the same integer.

Hash examples

- Some simple methods for creating keys from integers.
- Open DSA - Data Structures and Algorithms is a great reference.
- Modulo r

```
x % 16 # modulo 16
```

Hash examples

- Some simple methods for creating keys from integers.
- Open DSA - Data Structures and Algorithms is a great reference.
- Modulo r

```
x % 16 # modulo 16
```

- Binning (floor function or integer division)

```
x // 32 # need to know max(N) for r
```

Hash examples

- Some simple methods for creating keys from integers.
- Open DSA - Data Structures and Algorithms is a great reference.
- Modulo r

```
x % 16 # modulo 16
```

- Binning (floor function or integer division)

```
x // 32 # need to know max(N) for r
```

- Mid-Square method: square the value, use the middle digits in the hash

Hash considerations

- There are many choices for a hash function in practice. Considerations include:
- **Randomness**. For many applications (e.g. cryptography) we want no correlation between x and u.
- **Locality**. For other applications (e.g. locality sensitive hashing) we want similar x to produce similar u.
- **Collisions**. We may wish to reduce collisions on a subset of the potential input space. For example, if $x \in [0, r)$ and $u \in [0, r)$ it is possible to eliminate collisions.
- **Compute**. Hash functions vary in their compute cost.
- **Families**. It is often useful to be able to index a family of hash functions with the same computational cost that return different values.

Data Structures

- Data structures are representations of a **set** of data
- This representation is particularly important when sets are **dynamic**, i.e. grow or shrink
- We will perform **operations** on the set, which will have an associated computation cost
- The data structure has an associated space cost
- Making the right choice of data structure is an essential component of data science

Fixed size elementary data structures

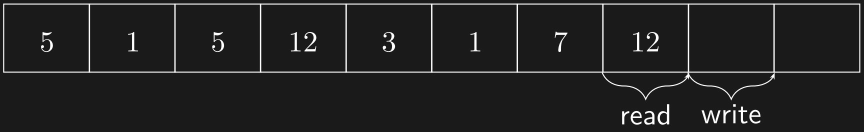
- We are familiar with the concepts of:
  - **Arrays**: A segment of memory containing n data of the same type
  - **Vectors**: Arrays with additional operations defined
  - **Multi-dimensional arrays**: Arrays of length $n = n_0 \times n_1 \times \cdots \times n_k$, with entries specified according to a protocol (e.g. row-wise)
  - **Matrices/Tensors**: Multidimensional arrays with additional operations defined
- It is clear that arrays are a fundamental concept!

Elementary data structures: Stacks and Queues

5	1	5	12	3	1	7	12		

- **Stacks**: Data are stored in an array using “first in, last out”: insertions and deletions occur at the same end
  - Implemented as a pointer to the last read location
- **Queues**: Data are stored in an array using “first in, first out”: insertions occur one end, deletions the other
  - Implemented as a pointer to the end (for writing) and start (for reading) that tracks removed items

Elementary data structures: Stacks and Queues



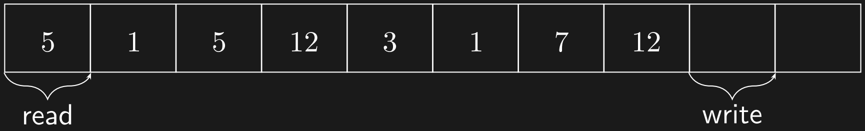
- **Stacks**: Data are stored in an array using "first in, last out": insertions and deletions occur at the same end
  - Implemented as a pointer to the last read location
- **Queues**: Data are stored in an array using "first in, first out": insertions occur one end, deletions the other
  - Implemented as a pointer to the end (for writing) and start (for reading) that tracks removed items

Elementary data structures: Stacks and Queues

5	1	5	12	3	1	7	12		

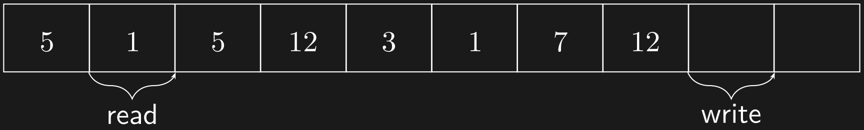
- **Stacks**: Data are stored in an array using "first in, last out": insertions and deletions occur at the same end
  - Implemented as a pointer to the last read location
- **Queues**: Data are stored in an array using "first in, first out": insertions occur one end, deletions the other
  - Implemented as a pointer to the end (for writing) and start (for reading) that tracks removed items

Elementary data structures: Stacks and Queues



- **Stacks**: Data are stored in an array using “first in, last out”: insertions and deletions occur at the same end
  - Implemented as a pointer to the last read location
- **Queues**: Data are stored in an array using “first in, first out”: insertions occur one end, deletions the other
  - Implemented as a pointer to the end (for writing) and start (for reading) that tracks removed items

Elementary data structures: Stacks and Queues



- **Stacks**: Data are stored in an array using "first in, last out": insertions and deletions occur at the same end
  - Implemented as a pointer to the last read location
- **Queues**: Data are stored in an array using "first in, first out": insertions occur one end, deletions the other
  - Implemented as a pointer to the end (for writing) and start (for reading) that tracks removed items

Elementary data structures: Stacks and Queues

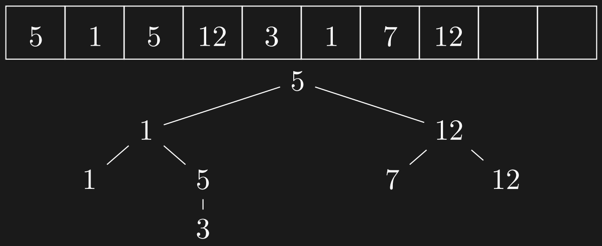
5	1	5	12	3	1	7	12		

- **Stacks**: Data are stored in an array using "first in, last out": insertions and deletions occur at the same end
  - Implemented as a pointer to the last read location
- **Queues**: Data are stored in an array using "first in, first out": insertions occur one end, deletions the other
  - Implemented as a pointer to the end (for writing) and start (for reading) that tracks removed items
- Despite implementation similarities, both have different Data Science properties!

Elementary data structures: Linked List

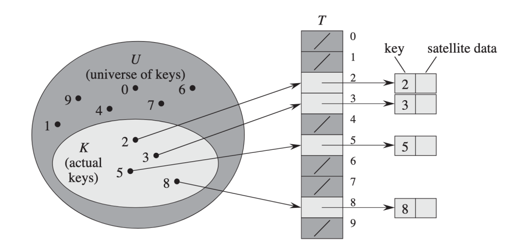
- **Linked list**: Data are stored in a list, with a pointer to the location of the next item
  - Fast traversion, insertion and deletion
  - Slow random access
  - Can be doubly linked

Elementary data structures: Binary Trees & Heaps



- **Binary Trees**: Data are stored in a **binary** linked list, i.e. each node has (up to) two children
  - Data can be stored at nodes or leaves
  - **Critical** to define the left/right operation!
- Position is decided by a key, which can be related to the value
  - In the picture, values $\leq x$ go left, $> x$ go right
  - Some binary tree structures assign values to internal nodes, e.g. means/ranges
- **Heaps**: A binary tree where each node's key is (larger) than it's children

Elementary data structures: Hash Tables



- **Hash Tables**: Data location determined by the **key**
- The key is a **hash** $x = h_l$: either of an attribute (e.g. a name), or of the value
- Advantage is $O(1)$ lookup cost. Usage is:
  1. Compute $u = h_2(x)$
  2. Set $u' = u\%r$
  3. To insert: store y at this position. On collision, we use some rule to find an empty space, such as rehashing, or storing a linked list.
  4. To lookup: retrive this value (using the same rule about collisions).

Signposting

- See 8.2 Part 2 on Algorithms for Data Science

References

- Data structures:
  - Cormen et al 2010 Introduction to Algorithms is very accessible and recommended for data structures.
  - Open DSA - Data Structures and Algorithms.