Topic: 4.1.3 Abstract Data Types (ADT)

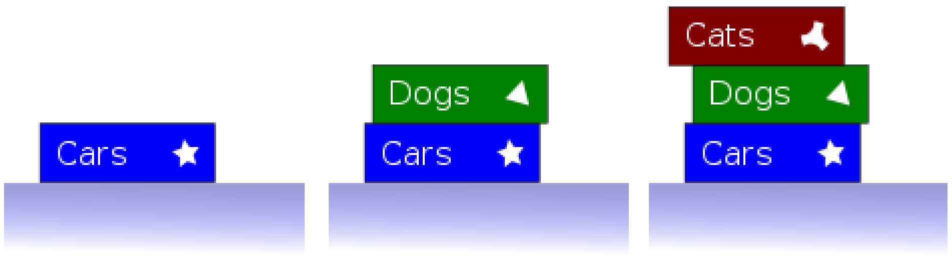
In Computer Science, an abstract data type (ADT) is a mathematical model for a certain data type or structures. This doesn't mean that the ADTs cannot be programmed. But that we must first understand them mathematically before we can implement them. ADTs are generally complex things that need functions and procedures to be programmed in order to get the right kind of operation for each ADT.

In the last section, you were introduced to several kinds of ADTs and the functions that can be performed on them. In general, ADTs are usually constructed using a built in data type. The most commonly reused data type is the ARRAY which can be used for stacks, queues, linked list, trees, etc…

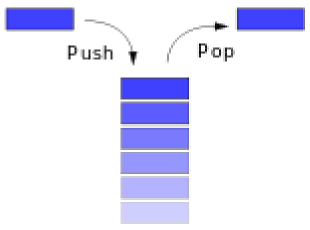
Stack:

A stack is an ADT that might involve a dynamic or static implementation. A stack is a **last-in-first-out** (LIFO) or **first-in-last-out** (FILO) ADT. Implementations should include two operations, **pushing** and **popping**, and a pointer to the **top of the stack**.

A real life example is a stack of books you might have on your desk:



In this example we keep adding (pushing) books to the stack. If we want to get at the bottom book about Cars, we must first remove (pop) all the books above it. Hence First In Last Out (FILO)



Let's take a look at a computer implementation of a stack:

- **Pushing:** Adds a new specified item to the top of the stack
- **Popping:** Removes the item from the top of the stack.

The algorithm for data retrieval in a stack is:

1. Starting from the head pointer, compare the item with the item you want.
2. If the items match, print "Match found at index (N)" and stop the searching.
3. Else if the items do not match, move the head pointer one index down then check that index.
4. Repeat steps 1 to 3 until you reach the end of the stack or until you find the data.
5. If you reach the end of the stack with no match found, print "No match found"

Topic: 4.1.3 Abstract Data Types (ADT)

For more details about the stack, and for algorithms to insert and delete an item in a stack, refer to chapter 4.1.2.

Queue:

A **queue** is a first-in first-out (FIFO) abstract data type that is heavily used in computing. Uses for queues involve anything where you want things to happen in the order that they were called, but where the computer can't keep up to speed. For example:

Printer Queue - you want print jobs to complete in the order you sent them, i.e. page 1, page 2, page 3, page 4 etc. When you are sharing a printer several people may send a print job to the printer and the printer can't print things instantly, so you have to wait a little while, but the items output will be in the order you sent them

Advantage: Fast to implement, with simple code.

Disadvantage: Uses up large amounts of memory, might overwrite important memory locations.

The algorithm for data retrieval in a queue is:

Starting from the head pointer, compare the item at that index with the item you want.

1. If the items match, print "Match found at index (N)" and stop the searching.
2. Else if the items do not match, move the head pointer one index down and check the data at that index.
3. Repeat steps 1 to 3 until you reach the end of the queue or until you find the data.
4. (The end of the queue will actually be when you keep decrementing the head pointer until it reaches the same level as the tail pointer!!!)
5. If you reach the end of the queue (head pointer = tail pointer) with no match found, print "No match found" and end the searching.

For more details about the queue, and for algorithms to insert and delete an item in a queue, refer to chapter 4.1.2.

Topic: 4.1.3 Abstract Data Types (ADT)

Linked List:

It is a dynamic abstract data type that uses pointers to vary memory used at run time.

A common question asked is why do we use linked lists when we can just use arrays?

There are a number of benefits and drawbacks of using a linked list over an array.

1. The memory used by a linked list can vary at run time, meaning that memory isn't a fixed size (unlike the array)

2. In an array, memory spaces are continuous (even on the hard drive). So when you want to initialize an array, there has to be enough continuous free spaces on your hard-drive, and if not, then you will have to reduce the size of your array. Linked lists help tackle this problem, one part of the list can be on one end of the hard-drive and another part of the list on the other end of the hard-drive, the linked list will simply link both lists and display it as if the data were continuous.

3. Linked lists, can keep a track of non-continuous data, but this will slow down the access to the data. Arrays have very fast access to data.

For more details about the linked list, and for algorithms to insert, delete, and search for items in a linked list, refer to chapter 4.1.2 pages 8, 9, and 10.

Binary Tree:

This topic has already been explained in depth in chapter 4.1.2.

Dictionary:

A dictionary is an abstract data type composed of a collection of (key, value) pairs, such that each possible key appears just once in the collection.

Operations associated with this data type allow:

- Addition of pairs to the collection
- Removal of pairs from the collection
- Lookup of the value associated with a particular key

A standard solution to the dictionary problem is a hash table. In some cases, it is also possible to solve the problem using binary search trees.

Dictionaries have many applications including many programming patterns such as **memorization** and the **decorator pattern**.

Topic: 4.1.3 Abstract Data Types (ADT)

Example

Suppose that the set of loans made by a library is to be represented in a data structure. Each book in a library may be checked out only by a single library patron at a time. However, a single patron may be able to check out multiple books. Therefore, the information about which books are checked out to which patrons may be represented by a dictionary in which the books are the keys and the patrons are the values.

For instance, in Python programming language, the current checkouts may be represented by a dictionary.

```
{
    "Great Expectations": "John",
    "Pride and Prejudice": "Alice",
    "Wuthering Heights": "Alice"
}
```

A lookup operation with the key “Great Expectations” in this array would return the name of the person who checked out that book, John. If John returns his book, that would cause a deletion operation in that dictionary, and if Pat checks out another book, which would cause an insertion operation, leading to a different state:

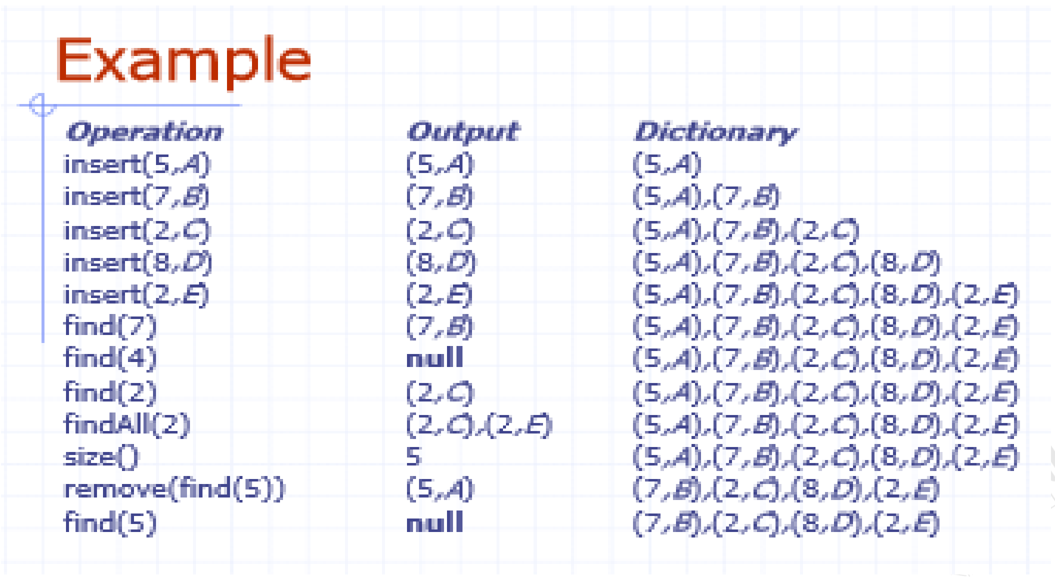
```
{
    "Pride and Prejudice": "Alice",
    "The Brothers Karamazov": "Pat",
    "Wuthering Heights": "Alice"
}
```

In this new state, the same lookup as before, with the key “Great Expectation”, would raise an exception because this key is no longer present in the array.

A dictionary could have the following methods:

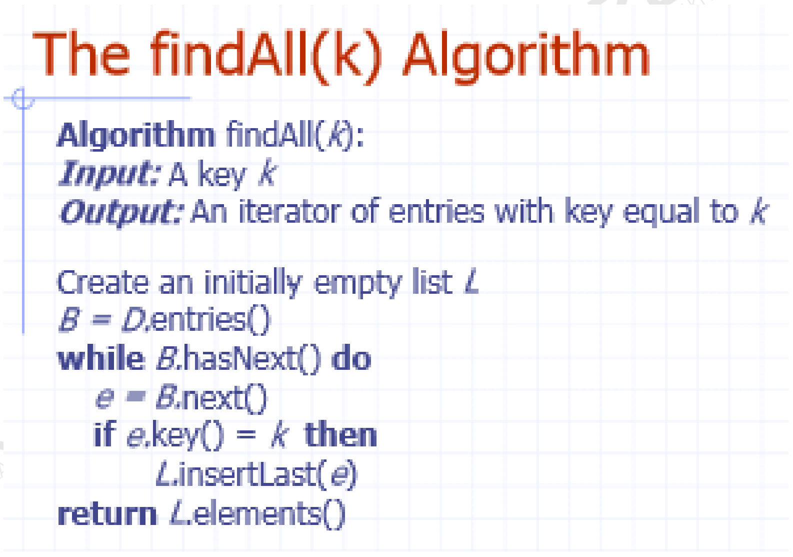
- find(k): if the dictionary has an entry with the key k, return it, else, return null.
- findAll(k): returns an iterator of all entries with key k.
- insert(k, o): inserts and returns the entry (k, O).
- remove(e): remove the entry e from the dictionary.
- size(): return the size of the dictionary
- isEmpty(): if the dictionary is empty, then this Boolean data type will return true value.

Topic: 4.1.3 Abstract Data Types (ADT)



Example

Operation	*Output*	*Dictionary*
insert(5,*A*)	(5,*A*)	(5,*A*)
insert(7,*B*)	(7,*B*)	(5,*A*),(7,*B*)
insert(2,*C*)	(2,*C*)	(5,*A*),(7,*B*),(2,*C*)
insert(8,*D*)	(8,*D*)	(5,*A*),(7,*B*),(2,*C*),(8,*D*)
insert(2,*E*)	(2,*E*)	(5,*A*),(7,*B*),(2,*C*),(8,*D*),(2,*E*)
find(7)	(7,*B*)	(5,*A*),(7,*B*),(2,*C*),(8,*D*),(2,*E*)
find(4)	**null**	(5,*A*),(7,*B*),(2,*C*),(8,*D*),(2,*E*)
find(2)	(2,*C*)	(5,*A*),(7,*B*),(2,*C*),(8,*D*),(2,*E*)
findAll(2)	(2,*C*),(2,*E*)	(5,*A*),(7,*B*),(2,*C*),(8,*D*),(2,*E*)
size()	5	(5,*A*),(7,*B*),(2,*C*),(8,*D*),(2,*E*)
remove(find(5))	(5,*A*)	(7,*B*),(2,*C*),(8,*D*),(2,*E*)
find(5)	**null**	(7,*B*),(2,*C*),(8,*D*),(2,*E*)

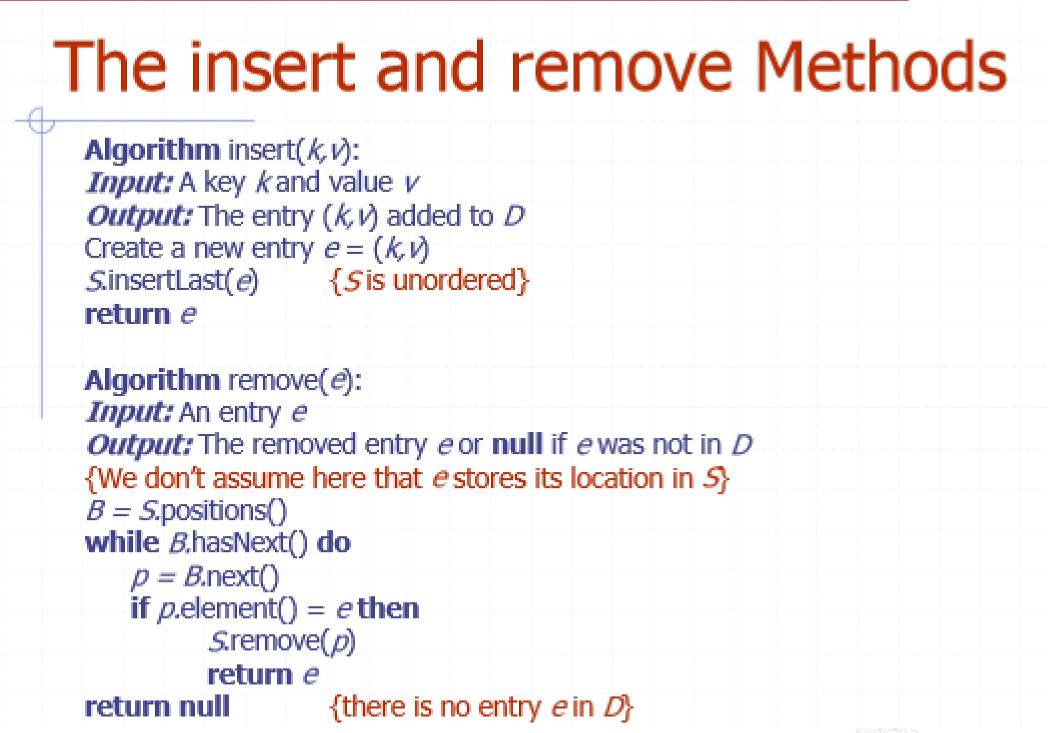


The findAll(k) Algorithm

```
Algorithm findAll(k):
Input: A key k
Output: An iterator of entries with key equal to k

Create an initially empty list L
B = D.entries()
while B.hasNext() do
    e = B.next()
    if e.key() = k then
        L.insertLast(e)
return L.elements()
```

Topic: 4.1.3 Abstract Data Types (ADT)



The insert and remove Methods

```
Algorithm insert(k,v):
Input: A key k and value v
Output: The entry (k,v) added to D
Create a new entry e = (k,v)
S.insertLast(e)          {S is unordered}
return e

Algorithm remove(e):
Input: An entry e
Output: The removed entry e or null if e was not in D
{We don't assume here that e stores its location in S}
B = S.positions()
while B.hasNext() do
    p = B.next()
    if p.element() = e then
        S.remove(p)
        return e
return null              {there is no entry e in D}
```