Programming, Data Structure and Algorithms in Python
Prof. Madhavan Mukund
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week - 06
Lecture - 04
Sets, Stacks, Queues

(Refer Slide Time: 00:01)

Data structures

- *Algorithms + Data Structures = Programs* Niklaus Wirth
- Arrays/lists — sequences of values
- Dictionaries — key-value pairs
- Python also has sets as a built in datatype

In the 1970s Niklaus Wirth, the inventor of the programming language Pascal wrote a very influential book called Algorithms plus Data Structures equals Programs. So, the title emphasises the importance of both algorithms and data structures as components of effective programs.

So far we have seen algorithms in some detail. So, now let us take a closer look at some specialized data structures. The data structures that we have seen that are built into python began with arrays and lists which are just sequences of values. We also saw dictionaries which are key value pairs and which are very useful for maintaining various types of information. Another built in data type that is available in python is the set.

(Refer Slide Time: 00:48)

Sets in Python

- List with braces, duplicates automatically removed

```
colours = {'red','black','red','green'}

>>> print(colours)
{'black', 'red', 'green'}
```

- Create an empty set

```
colours = set()
```

  - Note, not `colours = {}` — empty dictionary!

Set is like a list except that you do not have duplicates. In python, one way of writing a set is to write a list with braces like this. So, here we have associated with the name colours a list of values red, black, red and green. Notice that in setting it up, we have repeated red, but because this is a set, the duplicate red would be automatically removed. So, if we print the name colours, we just get the list black, red and green. Now, since the empty brace notation is already used, for empty dictionary if we want to create an empty set, we have to call the set function as follows.

(Refer Slide Time: 01:33)

Sets in Python

- Set membership

```
>>> 'black' in colours
True
```

- Convert a list into a set

```
>>> numbers = set([0,1,3,2,1,4])
>>> print(numbers)
{0, 1, 2, 3, 4}

>>> letters = set('banana')
>>> print(letters)
{'a', 'n', 'b'}
```

So, we say colours equal to set with no arguments. Like lists and other data structures, we can test membership using in. So, if in the previous lists set colours which had red, black and green, we ask whether black is in colours by using the word in, then, the return value is true. In general we can convert any list to a set using the set function.

We saw that if we give no arguments to set you get an empty set, but if we give a list such as this 1, 3, 2, 1, 4 with duplicates and assign it to the name numbers, then because its a set the duplicate ones will be removed and we will get a list of, we will get a set of numbers 0, 1, 2, 3, 4. Notice again that the order in which the set is printed need not be the order in which you provided it. This is very much like a dictionary sets; are optimized for internal storage to make sure there are no duplicates etcetera.

So, we should not assume anything about the order of elements in set. An interesting feature is that a string itself is essentially a list of characters. So, if we give a string to a set, then it produce the set function, then it produces a set which consists of individual letters from this set. So, if we give this string banana to the set function, then we get the three individual letters a, n and b without duplicates in the set.

(Refer Slide Time: 02:58)

Set operations

```
odd = set([1,3,5,7,9,11])
prime = set([2,3,5,7,11])
```

- Union
  `odd | prime → {1, 2, 3, 5, 7, 9, 11}`
- Intersection
  `odd & prime → {3, 11, 5, 7}`
- Set difference
  `odd - prime → {1, 9}`
- Exclusive or
  `odd ^ prime → {1, 2, 9}`

So, as you would expect sets support basic operations like their counterpart in mathematics, so suppose we set up the odd numbers to be the set of all odd numbers between 1 and 11and the prime numbers to be the set of all prime numbers from 1 and 11 between 2 and 11 using these set function as we saw before. If we write this vertical bar, then we can get the union of the two sets.

So, odd union prime will be those elements which are either in odd or in prime. So, we get one from the top two from the bottom 3, 5, 7, 9, 11. We get all the elements in both the sets, but without any duplicates. If we ask for the intersection of two sets, we use ampersand to denote this. We get those which occur in both sets, those sets, those numbers which are both odd and prime and in this case 3, 5, 7 and 11.

Notice again that the order in which these numbers are printed may be arbitrary. Set difference asks for those elements that are in odd, but not in prime. In other words, odd numbers that are not prime, in this particular collection 1 and 9 are examples of odd numbers that are not prime.

And finally, unlike union which collects elements which are in both sets, we can do an exclusive or which takes elements which are exactly in one of the two sets. If we use this

carrot symbol, then we will get 1 from the first set, 9 from the first set and 2 from the second set because 3, 5, 7, and 11 occur in both sets. So, we will not talk much more about sets, but you can use them in various contexts in order to keep track of a collection of values without duplicates using these built in operations.

(Refer Slide Time: 04:40)

Stacks

- Stack is a last-in, first-out list
  - push(s,x) — add x to stack s
  - pop(s) — return most recently added element
- Maintain stack as list, push and pop from the right
  - push(s,x) is s.append(x)
  - s.pop() — Python built-in, returns last element

Let us look at different ways in which we can manipulate sequences. A list as we saw is a sequence in which we can freely insert and delete values all over the place. Now, if we impose some discipline on this, we get specialized data structures one of which is a stack. A stack is a last in first out list. So, we can only remove from a stack the element that we last added to it.

Usually this is denoted by giving two operations. When we push an element on to a stack, we add it to the end of the stack and when we pop a stack, we implicitly get the last value that was added. Now, this is easy to implement using built in python list. We can assume that stacks grow to the right. So, we push to the right and we pop from the right. So, push s x would just be append x to s.

So, you can use the built-in append function that is available for lists to say s dot append x when we want to push and it turns out that python's lists actually have a built in

function called pop which removes the last element and returns it to us. So, we just have to say s dot pop, where s is a list and we get exactly the behavior that we expect of our stack.

(Refer Slide Time: 05:57)

Stacks

- Stacks are natural to keep track of recursive function calls
- In 8 queens, use a stack to keep track of queens added
  - Push the latest queen onto the stack
  - To backtrack, pop the last queen added

A stack is typically used to keep track of recursive function calls where we want to keep going through a sequence of functions and then, returning to the last function that was called before this. In particular when we do back tracking, we have a stack like behavior because as we add queens and remove them, what we need to do effectively is to push the latest queen onto the stack, so that when we backtrack, we can pop it and undo the last move.

(Refer Slide Time: 06:29)

Queues

- First-in, first-out sequences
  - `addq(q,x)` — adds x to rear of queue q
  - `removeq(q)` — removes element at head of q
- Using Python lists, left is rear, right is front
  - `addq(q,x)` is `q.insert(0,x)`
    - `l.insert(j,x)`, insert x before position j
  - `removeq(q)` is `q.pop()`

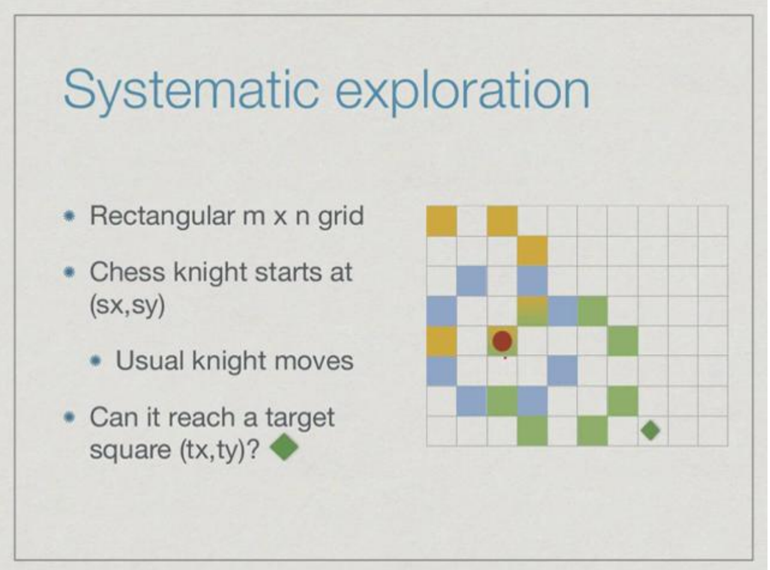
Another disciplined way of using a list is a queue. Unlike a stack which is last in first out, a queue is a first in first out sequence. In other words, we add at one end and we remove at another end. This is exactly like a queue that you see in real life, where you join the queue at the back and when your turn comes, you are at the head of the queue and then you get served. So, add q will add x to the rear of the queue and remove q will remove the element which is at the head of the q.

Once again we can use python lists and it turns out that it is convenient to assume that a list it that represents a queue has its head at the right end rather than the rear at the left and the head at the right. This is because we can use pop as before, but now when we want to insert into a queue, we can use the insert function that is provided with this. We have not seen this explicitly, but if you have gone through the documentation, you will find it.

If I have a list l and if I insert with two arguments j and x, what it means is to put the value j before position j, put the value x before position j in particular if I insert at position 0, this has the effect of putting something before every element in the list. So, add q q comma x is just the same as q dot insert 0 comma x.

In other words, push an x to the beginning. If I have a queue at this form which has some values v 1, v 2 and so on, then this insert function will just put an x at the beginning and as we said before, the reason we have chosen to use this notation is that we can then use the pop to just remove the last element of the list. Queues and stacks can both be like easily implemented using built-in lists.

(Refer Slide Time: 08:22)



So, one typical use of the queue is to systematically explore through search space. Imagine that we have a rectangular m cross n grid and we have a knight. Knight as a chess piece starting at a position s x comma s y. In this case, the knight is denoted by this red symbol. So, this is our knight. Now, the knight move, if you are familiar with chess is to move two squares up and one square left. This is a knight move.

Similarly, this is a knight move; similarly this is a knight move and so on. So, knight move consists of moving two squares in one direction, then one square across. So, these are all the positions that are reachable from this initial position, where the knight move there are eight of them. So, our question is that we have this red starting square and we have a green diamond indicating a target square.

Can I hop using a sequence of knight moves from the red square to the green diamond?

So, one way to do this is to just keep growing the list of squares one can reach. So, in the first step we examine these 8 squares that we can reach as we said using one move from the starting position and we mark them as squares that are available to us to reach in one step. Now, we can pick one of them for instance one of the top left and explore what we can reach from there. So, if we start at this square for instance and now we explore its neighbors, some of its neighbors are outside the grid. So, we throw them away. We keep only those neighbors inside the grid and one of them notice brings us back to the place where we started from.

Now, we could pick another square for example, we could pick this square over here and if we explore that it will again in turn produce 8 neighbors and some of these neighbors overlap the yellow neighbors. I indicate it by joint shading of yellow and green and in particular because both of them were originally reached from the starting point.

Of course, the starting point reaches from both of them. The starting point is both colored yellow and green. So, as you can see in the process of marking, these squares, sometimes we mark the square twice and we have to have a systematic way of making sure that we do this correctly and do not get into a loop.

(Refer Slide Time: 10:34)

Systematic exploration

- X1 — all squares reachable in one move from (sx,sy)
- X2 — all squares reachable from X1 in one move
- . . .
- Don't explore an already marked square
- When do we stop?
  - If we reach target square
  - What if target is not reachable?

So, what we are trying to do is the following. So, in the first step we are trying to mark all squares reachable in one move from the starting point s x comma s y. Then, we try to mark all squares reachable from x 1 in one move, call this x 2, and then we will explore all squares reachable from x 2 in one move, call this x 3 and soon.

Now, one of the problems is that we saw that since we could reach x2 from x 1in one move, then the squares that can reach from x 2 will include squares in x 1. So, how do we ensure that we do not keep exploring an already marked square and go around and round in circles and related to this question is how do we know when to stop.

(Refer Slide Time: 11:40)

Systematic exploration

- Maintain a queue Q of cells to be explored
- Initially Q contains only start node (sx,sy)
  - Remove (ax,ay) from head of queue
  - Mark all squares reachable in one step from (ax,ay)
  - Add all newly marked squares to the queue
- When the queue is empty, we have finished

Of course since we know that we are looking for the target square, if ever we marked the target square, we can stop. On the other hand, it is possible that the target square is not reachable. In this case, we may keep going on exploring without ever realizing that we are fruitlessly going ahead and we are never going to reach the target square. So, how do we know when to stop? So, a queue is very useful for this. What we do is we maintain at any point a queue of cells which remain to be explored. Initially the queue contains only the start node which is s x comma s y.

At each point we remove the head of the queue and we explore its neighbors, but when

we explore its neighbors, we mark these neighbors. Some of them may already be marked. So, we look at a x, a y, the element we remove from the head of the queue and we look at all the squares reachable at one step.

So, reachable means I can take one knight move and go there and the result of this knight move does not take me off the board. So, I mark all these squares which are reachable from a x and a y, some of which were already marked, some of which are marked just now. So, what I do is, I take the ones which I have newly marked and add them to the queue saying that these are being newly marked.

Now I need to also explore these squares for what I can reach from there. So, this guarantees that a square which has been reached once will never be reintroduced into the queue. Finally, we keep going until the queue is empty. When the queue is empty, there have been no new squares added which are unmarked before they were added. So, there is nothing more to explore and we have gone to every square we can possibly visit.

(Refer Slide Time: 13:04)

Systematic exploration

```
def explore((sx,sy),(tx,ty)):
  marked = [[0 for i in range(n)]
                    for j in range(m)]
  marked[sx][sy] = 1
  queue = [(sx,sy)]
  while queue != []:
    (ax,ay) = queue.pop()
    for (nx,ny) in neighbours((ax,ay)):
      if !marked[nx][ny]:
        marked[nx][ny] = 1
        queue.insert(0,(nx,ny))
  return(marked[tx][ty])
```

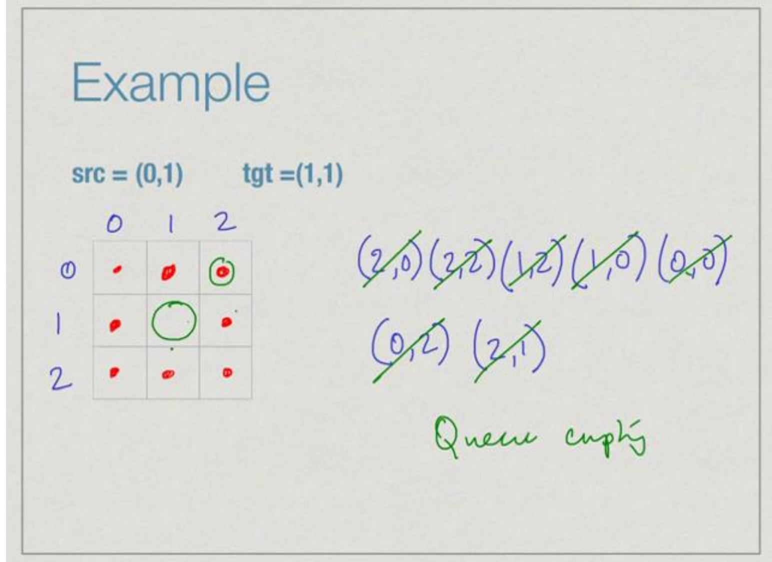
Here is some python pseudo code for this. We are going to explore from s x, s y to t x, t y. We assume that we have given to us the values m and n indicating the number of rows and columns in our grid. So, what we do is initially we set the marked array to be 0.

Remember this list comprehension notation. It says 0 for i in range n gives us a list of n zeros and we do this m times for j in range m. So, I we will get a list consisting of m blocks and each block having n zeros. This says that initially nothing is marked.

Now, we set up the thing by saying that we mark the starting node and we insert the starting node from the queue. Now, so long the queue is not empty, we pop one element from the queue. In this case s x, s y will come out. Now, there is a function which we have not written, but which will examine all the neighbors that I can reach from a x, a y and give me a list of such pairs of nodes I can reach.

For each neighbor nx ny if it is not marked, then I will mark it and I will insert it into the queue, right. So, I pull out an element from the queue to explore, look at all its neighbors those which are not marked, I mark and put them back in the queue and finally, in this case I am not even going to check whether I have marked t x or t y in the middle. I know that if I have a finite set of squares at some point, this process has to stop. At the end I will return whether I will return the value of marked at the target node t x, t y. So, if I have reached it, this will return 1 which is true. If it is not reached, it will return 0 which is false.

(Refer Slide Time: 14:53)



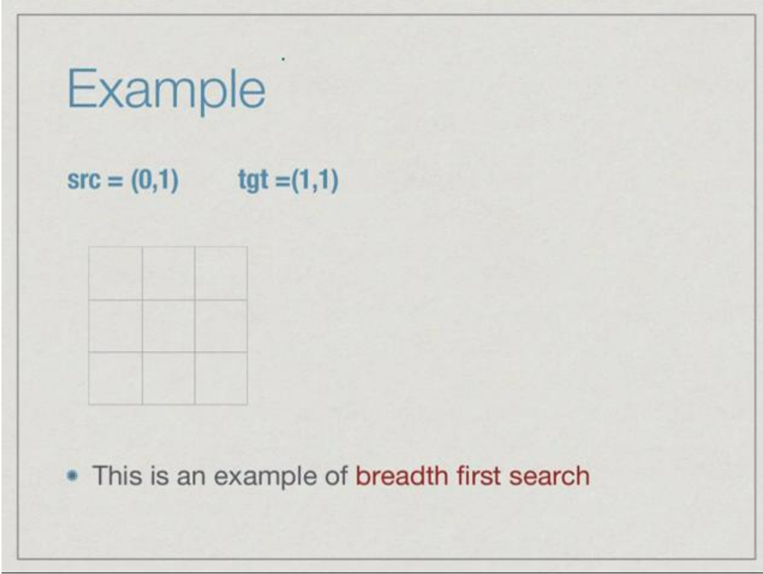
Let us look at an example of how this works. So, here we have a three by three grid. Remember that the cells are 1 0 1 2 and 0 1 2 by our naming convention we want to start from the top center square. This is our source and here in the center is our target. So, let us erase all these marks and set up this thing as we expect. So, we say that initially the queue that we want to have as a source node and we mark the source node in the grid. The marking is indicated by a red mark. So, this is how we start.

So, our first step is to remove this from the queue and explore its neighbors. Now, its neighbors are 2, 0, and 2. This means we will henceforth remove these brackets because it is more annoying. So, you just grow it like this. So, we say that my queue consists of 2 comma 0 and 2 comma 2. This is my queue of vertices way to be explored. At each step I will now remove the first element of the queue and explore its neighbors. When I explore the neighbors of 2 0, I will find one of them is of course is where I start it from. I only look at unmarked neighbors. So, an unmarked neighbor is 1 2. I will add that back at the end of the queue.

Now, proceeding I will take the next element of the queue which is 2 2 and look at its neighbors. So, 2 2 can go back again to the original thing and it also has a new thing here which is 1 0. So, continuing like this I remove 1 2 which is this one and then, look at its neighbors. So, one of its neighbors is 2 0, but one of them is 0 0. So, I get a new neighbor 0 0 here and then, I continue by taking 1 0 of the queue. So, 1 0 is this one. So, it has one new neighbor unexplored which is that one. So, my queue now has 0 0 followed by 0 2. Then, when I explore 0 0, I get this neighbor at the bottom which is 2 1.

Now, when I remove 0 2 which is this one, I find that both these neighbors I explored. So, I add nothing. I continue with 2 1. Again, I find both its neighbors explored and do nothing. Now at this point, the queue is empty and since the queue is empty, I stop and I find that my square of interest namely 1 1 was not marked. Therefore, in this case the target is not reachable from the source node.

(Refer Slide Time: 17:35)



This is actually an example of breadth first search which you will study if you look at graphs, but it just illustrates that a queue is a nice way to systematically keep track of how you explore through a search space.

(Refer Slide Time: 17:47)

Summary

- Data structures are ways of organising information that allow efficient processing in certain contexts
- Python has a built-in implementation of sets
- Stacks are useful to keep track of recursive computations
- Queues are useful for breadth-first exploration

To summarize data structures are ways of organizing information that allow efficient

processing in certain contexts. So, we saw that python has a built-in implementation of lists of sets rather we also saw that we can take sequences and use them in two structured ways. So, stack is a last-in first-out list and we can use this to keep track of recursive computations and queues are first-in first-out list that are useful for breadth first exploration.