Nodes: A Basis for Implementing Linear Data Structures

CMPT 145

Copyright Notice

Learning Objectives

After studying this chapter, a student should be able to:

- To describe the concept and structure of a node.
- To explain the operations of the Node ADT.
- To employ Node ADT operations in Python programs.

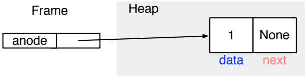
Motivation

- Python lists are very useful for programmers.
  - Easy for novices to learn.
  - Very practical for many applications.
- Python lists are based on fixed length blocks of memory.
  - You will study this idea in CMPT 214 (C/C++ arrays).
- It's educational to consider alternatives.
  - In CMPT 145 we study node-chains and linked lists.
- We study these ideas because:
  - Very good programming practice
  - Deepen your understanding of Python
  - Broaden your understanding of computer science

Data Structure: Node

A node is a very simple object:

```
anode = node(1, None)
```



It stores 2 values only.

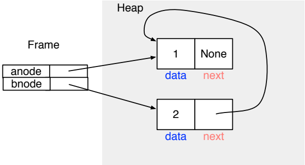
1. Any data value.
2. A reference to another node (or `None`)

The Node ADT is not built into Python. Use the module provided by CMPT 145.

Nodes create chains of data values

Two nodes, linked together:

```
anode = node(1, None)
bnode = node(2, anode)
```



We use the second argument to refer to another node.

Node ADT

- Purpose:
  - Building block for data sequences.
- Implementations:
  - Object with 2 attributes
    1. A data value
    2. A reference to another node (or `None`)
- Operations:
  - Create a node
  - Set the data value for a given node
  - Set the reference to the next node for a given node
  - Return the data value of a given node
  - Return the reference to the node of a given node

Code Walk Through

Python keyword arguments

- Normal function parameters are based on position.

```python
def fun3(a, b, c):
    pass
fun3(1,2,3)
```

- keyword arguments use the parameter name:

```python
def fun2(a, b, c=0):
    pass
fun2(1,2,c=3)
```

- The assignment in the parameter list establishes a default value

Python keyword arguments

- You only need to give a value if you want something other than the default:

```
fun2(1,2,c=3)  # ignore the default value
fun2(1,2)      # use the default value
```

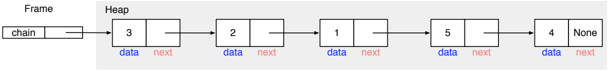
- Position based parameters must precede keyword arguments in the definition.

Analogy: Nodes are freight cars

- A node object is like a freight car in a railroad train.
- Each node can contain some cargo (data)
  - The data can be any kind of value.
  - We will keep it simple in our examples.
- Each node points to a node that comes after it (next)
  - We must take care to use this attribute only for another node.

Analogy: Node chains are trains

- A node chain is like a railroad train.
  - Each node is like a freight car.
- A node points to the next node in the chain.
  - By design, a node does not know what's in front of it.



- Each node can contain some cargo (data)
- A variable that knows the first node in a chain is called the anchor.
- The last node in the chain must have `None` stored as its next value.

Common questions 1

- Can we create a chain that has a loop back to the beginning?

  This is useful in some applications, but confusing for beginners. We won't study them in CMPT 145.

- Can we create a different kind of node that points backwards and forwards?

  This is useful in some applications, but confusing for beginners. We won't study them in CMPT 145.

Common questions 2

- Is this how Python lists work?

  No. Python lists are based on fixed length blocks of memory. This is a design decision based on a compromise. Python lists are good at some things, but not the best for every application.

- Why are we studying node chains?
  - Node chains are better than Python lists for some applications!
  - We need this idea for Chapters 16, 17, 20-23.
  - You will study more advanced ideas in CMPT 280.

Example 1

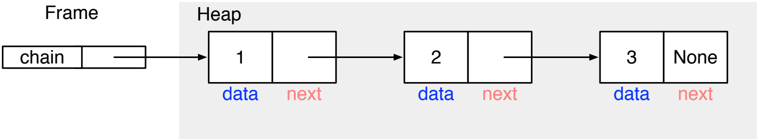
Draw a diagram for the following code sequence:

```
x = N.node(5, None)

y = N.node(1, x)

z = N.node(8, y)

print(x.get_data())
print(z.get_next().get_data())
print(z.get_next().get_next().get_data())
```

You cannot do this reliably in your head. Draw a diagram.

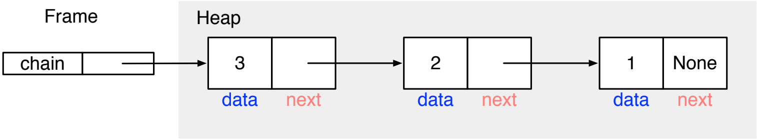
Example 2

Write the code to produce the following sequence:



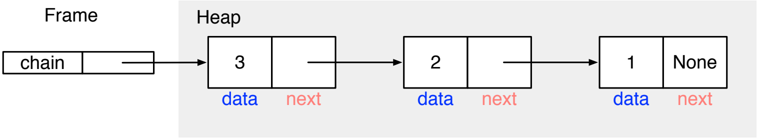
Example 3

Write the code to produce the following sequence:



Example 4

Given the following sequence:

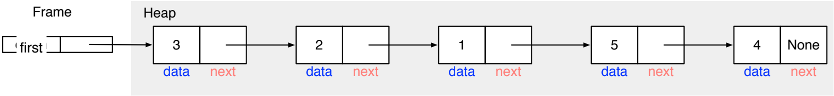


Write a print statement using the above chain that:

1. Uses the data in the chain to evaluate to 1
2. Uses the data in the chain to evaluate to 6
3. Uses the data in the chain to evaluate to 9

Simple algorithms on Node records

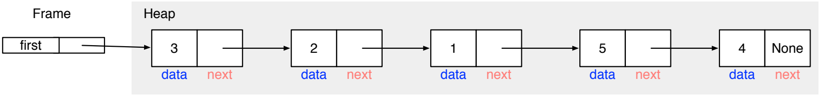
Suppose the variable first is a reference to the first node in the sequence:



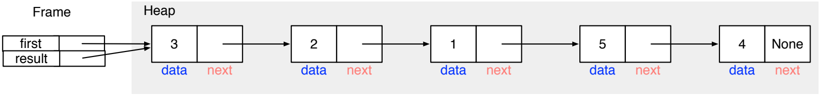
Use the Node ADT to:

1. Remove the 3 from the sequence
2. Add a new value 6 at the beginning of the sequence
3. Add a new value 7 at the end of the sequence

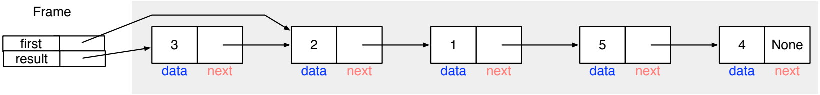
Removing 3 from the sequence



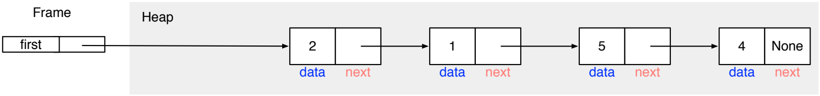
Removing 3 from the sequence



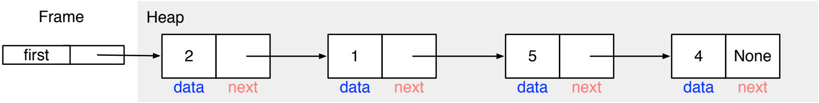
Removing 3 from the sequence



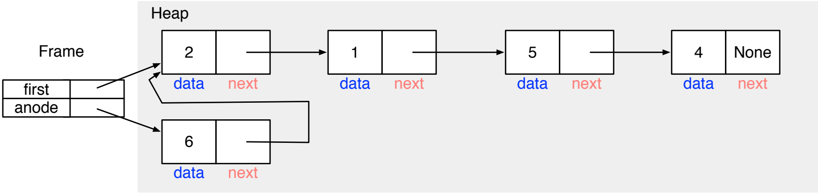
Removing 3 from the sequence



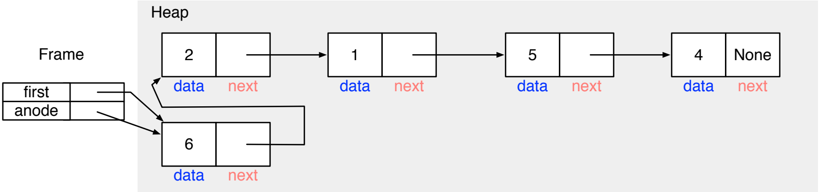
Add 6 at the beginning



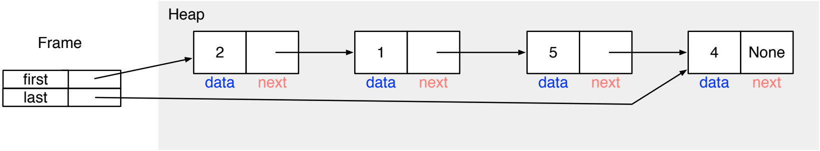
Add 6 at the beginning



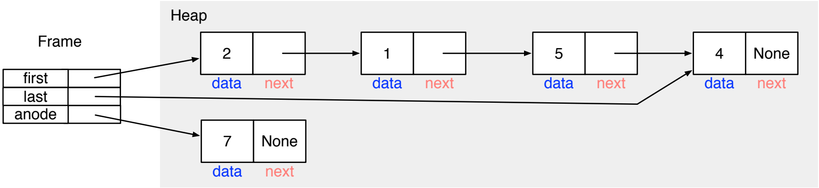
Add 6 at the beginning



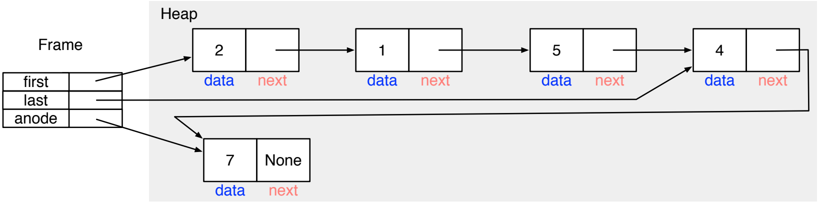
Add 7 at the end



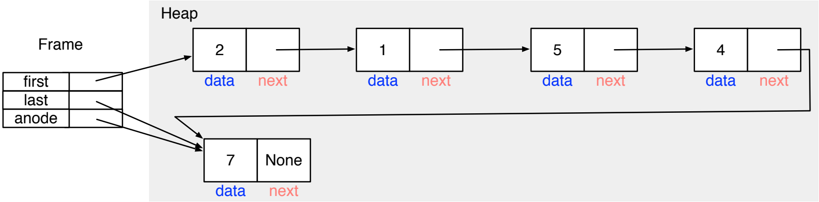
Add 7 at the end



Add 7 at the end



Add 7 at the end



Simple algorithms on Node records

- Nodes chains can have any number of nodes: 0, 1, 2, …
- Many algorithms on node chains require a loop. e.g.,
  - Count the number of nodes in a node chain
  - Print the data values in the node chain
  - Does the node chain contain the data value 4?
  - Replace every occurrence of the value 4 with the value -4.
  - Add the data value to the chain after the value 4.
- Loops on node chains have a common pattern!

Simple walking loop

This loop takes one step along the node chain, stopping at the end of the chain.

```python
chain = ...
walker = chain
while walker is not None:
    # do something with walker
    walker = walker.get_next()
```

- We use a `walker` to step along the node chain
- If we change `chain` we are moving the anchor point!
- After the loop is over, `walker` has jumped of the end of the node chain
- This is useful when the algorithm works exclusively `walker`.

Lookahead loop

This loop takes one step along the node chain, stopping on the last node in the chain:

```
chain = ...
walker = chain
while walker.get_next() is not None:
    # do something with walker and walker.get_next()
    walker = walker.get_next()
```

- After the loop is over, `walker` remains on the last node in the node chain.
- This is useful when the algorithm has to work on `walker`, and the node after it.

Double-walker loop

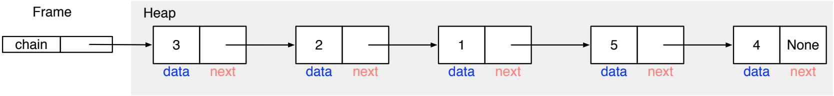
This loop takes one step along the node chain, remembering the node in front of it.

```
chain = ...
walker = chain
previous = None
while walker is not None:
    # do something with walker and previous
    previous = walker
    walker = walker.get_next()
```

- After the loop is over, `walker` remains on the last node in the node chain.
- This is useful when the algorithm has to work on `walker`, and the node before it.

Simple algorithms on Node records

Suppose the variable `chain` is a reference to the first node in the sequence:



Use the Node ADT to:

1. Count the number of nodes in the node chain.
2. Display all numbers in the chain
3. Does the node chain contain the value 4?
4. Remove the 4 from the sequence
5. Change the list so that 5 follows 2 (“delete 1”)

The Node ADT

- A simple data structure, hidden behind an interface.
- Chaining nodes together creates a sequence.
- Stacks and queues can be implemented using nodes.
- Nodes are seriously valuable!