Data Structure Examples

C++
ArrayDemo.cpp -> p. 4
StringDemo.cpp -> p. 7
Struct.h -> p. 9
StructDemo.cpp -> p. 10
Node.h -> p. 12
  LinkedList.cpp -> p. 13
  LinkedListDemo.cpp -> p. 18
Stack.h -> p. 20
StackDemo.cpp -> p. 22
Queue.h -> p. 25
  QueueDemo.cpp -> p. 26
Set.h -> p. 30
  SetDemo.cpp -> p. 32
  ArrayList.h -> p. 34
    ArrayListDemo.cpp -> p. 36
Map.h -> p. 38
  MapDemo.cpp -> p. 40

Python
Node.py -> p. 42
  LinkedList.py -> p. 43
    LinkedListTest.py -> p. 46
Stack.py -> p. 49
  StackTest.py -> p. 50
Queue.py -> p. 53
  QueueTest.py -> p. 54
UserSet.py -> p. 56
  SetTest.py -> p. 58
    TupleDemo.py -> p. 60
Map.py -> p. 61
  DictDemo.py -> p. 63
    GraphDemo.py -> p. 66
Java

ArrayDemo.java ————> p. 67
Node.java ———————————> p. 70
  LinkedList.java ———> p. 71
  ListDemo.java ————> p. 74
Stack.java ——————————> p. 75
  StackDemo.java ————> p. 77
Queue.java —————————-> p. 79
  QueueDemo.java ————> p. 81
UserSet.java ————————> p. 82
  UserSetTest.java ———> p. 83
  UserArrayList.java -> p. 84
  ArrayListDemo.java -> p. 86
Dictionary.java ————> p. 88
  DictionaryDemo.java > p. 90
TreeNode.java ————> p. 91
  BinaryTree.java ———> p. 92
TreeDemo.java ————> p. 94
Author’s Note

Thank you very much for your purchase and your interest in expanding your programming knowledge. “Data Structure Examples” offers 50 concrete examples and demos to help you gain better understanding of data structures. “Data Structure Examples” uses three popular programming languages (C++, Python, and Java) in order to teach you how to write various data structures using different languages. This gives you the preparation needed to start creating and using your own data structures, in no time.

Common data structures that are written in all three languages include: Linked Lists, Stacks, Queues, and Maps. Demos are then used to use our self-made data structures as well as data structures present in the various languages. More data structures are present in “Data Structure Examples” than the before mentioned, however those are present in each language and are fundamental.

Prior programming experience is required in order to understand these examples, so if you need to check out my other programming example books: “C Examples”, “C++ Examples”, “Python Examples”, or “Java Examples”, please do so. Please do note that this book stresses data structures, not algorithms. Please type out the code in order to get the most out of this book, take your time and have fun. There is much to learn so let us begin.

Thank You.
#include <iostream>

/*
Author | Date | Email
Arrays are statically implemented data structures by some
programming languages like C and C++. Array size in a static program
must be known at compile time and cannot be altered at run time. In dynamic programming
languages arrays are objects and give the
programmer a way to alter the array size at run time.
*/

int main(){

    //we must know the size prior to run time
    int array[5];
    for(int i = 0; i < 5; i++){
        array[i] = i*i;
        printf("array[%d]: %d\n", i, i*i);
    }

    //in order to add to array, we can use
    //individual indices
    array[0]=100;//was 0
    printf("array[0]: %d\n",array[0]);

    //in order to resize array we must create a new array
    int arrayTwo[10];
    for(int i = 0; i<10; i++){
        arrayTwo[i] = (i<5) ? array[i]:i*2;
        printf("arrayTwo[%d]: %d\n",i, arrayTwo[i]);
    }

    /*
    Arrays are "easy" to implement, access, and delete members.
    As such, they are the most widely used data structure. However,
    if you want to insert an element at an already given position,
    you must create a new array one size bigger, shift everything,
    and initialize the given index. This is a time consuming task,
    especially if the array is of large size. The same is true about
    deleting a member.
    */

    //we want to add 7 to "array", at index 3, without
    //altering the rest of the members
    int arrayThree[6];
    for(int i = 0; i < 6; i++){
        if(i == 3)
arrayThree[i] = 7;
else
    arrayThree[i] = (i<3) ? array[i] : array[i-1];
printf("arrayThree[%d]: \%d\n", i, arrayThree[i]);
}

/*
 Searching for a member in an unsorted array, can at worst take 
 O(n) running time to finish, where n is the size of the array. 
 However if array is sorted, you can use logic to find members 
 more quickly. In this case O(logN) is the worst case running time. 
 */

//we want to see if 9 is in our array 
//since we know our array is sorted we 
//can deal with it as such 
int target = 9;
int size = sizeof(array)/sizeof(int);
int middle = (int)(size/2);
bool b = array[middle] > target;
int index = -1; //we will init to -1 so 
//if this returns we will know we didn’t find it
if(b){
    for(int i = 0; i <= middle;i++)
        if(array[i] == target)
            index = i;
}
else{
    for(int i = middle; i < size; i++)
        if(array[i] == target)
            index = i;
}
printf("Index where target was found: \%d\n",index);

return 0;
Output:

array[0]: 0
array[1]: 1
array[2]: 4
array[3]: 9
array[4]: 16
array[0]: 100
arrayTwo[0]: 100
arrayTwo[1]: 1
arrayTwo[2]: 4
arrayTwo[3]: 9
arrayTwo[4]: 16
arrayTwo[5]: 10
arrayTwo[6]: 12
arrayTwo[7]: 14
arrayTwo[8]: 16
arrayTwo[9]: 18
arrayThree[0]: 100
arrayThree[1]: 1
arrayThree[2]: 4
arrayThree[3]: 7
arrayThree[4]: 9
arrayThree[5]: 16
Index where target was found: 3
StringDemo.cpp

#include <iostream>
#include <string>

/*
   Author | Email | Date
   Strings are sequences of characters, and are traditionally understood as a data type rather than a data structure. Strings, depending on the language, are either allocated memory statically or dynamically. What this means is that as far as memory is concerned and how data is stored in memory, Strings are data structures. Statically allocated memory means the memory space is fixed and the String cannot go beyond a certain size. Dynamically allocating memory space means the String can change size during run time. In C/C++ Strings are allocated memory space statically therefore they can not change size during run time, unless we use the following technique.
*/

int main(){
// a string is an array of chars
char name[] = {'T','u','t','o','r','i','a','l','\0'};
for(int i = 0; i < sizeof(name)/sizeof(char); i++){
    printf("name[%d]: %c\n", i, name[i]);
}

// because strings are an array of characters we can use the
// techniques in the "ArrayDemo.cpp" page
// in order to resize, search, access, delete, and add members.

/*
  This is all well and good, but now let us see how to dynamically
  allocate memory space, in order to not waste memory space, nor
  cause an error by trying to insert more characters than allowed.
*/
// malloc returns a void pointer so we must cast the pointer we want
char* name2 = (char*)malloc(sizeof(char));
char input[] = {'u','n','k','n','o','w','n','\0'};
for(int i = 1; i < sizeof(input)/sizeof(char); i++){
    // each time the for loop "grows", so does the size of the array
    // this technique can be applied to user input of unknown length
    name2 = (char*)realloc(name2,sizeof(char)*(i));
    name2[i-1]=input[i-1];
}
for(int i = 0; i < sizeof(name2)/sizeof(char); i++){
    printf("name2[%d]: %c\n", i, name2[i]);
}
return 0;
}

Output:
name[0]: T
name[1]: u
name[2]: t
name[3]: o
name[4]: r
name[5]: i
name[6]: a
name[7]: l
name[8]:
name2[0]: u
name2[1]: n
name2[2]: k
name2[3]: n
name2[4]: o
name2[5]: w
name2[6]: n
name2[7]:

Struct.h
typedef struct{
  string name;
  int age;
  string phoneNumber;
} Person;
```cpp
#include <iostream>
#include "Struct.h" //notice the import statement

/*
  Author | Email | Date
  StructDemo: Exercises the header file “Struct.h”. Please notice the import statement. How to use data structures, and how they are different from arrays. To dereference a pointer to a struct we need to use the -> operator.
*/

int main()
{

  //now we'll start initializing our structures
  Person person;
  person.name = "Tutorial Series";
  person.age = 6;
  person.phoneNumber = "555-555-0000";

  Person person2;
  person2.name = "Frank Sinatra";
  person2.age = 15;
  person2.phoneNumber = "555-555-1111";

  Person person3;
  person3.name = "Bob Ross";
  person3.age = 30;
  person3.phoneNumber = "555-555-2222";

  //now we will create a phone book representation
  //to store our Person structures.
  //Can you modify this to make the phoneBook grow
  //and shrink as needed?
  Person* phoneBook = (Person*)(malloc(sizeof(Person)*3));
  phoneBook[0]=person;
  phoneBook[1]=person2;
  phoneBook[2]=person3;

  for(int i = 0; i < 3; i++){
```
std::cout<<"Person "<<i+1<<": 
";
std::cout<<phoneBook[i].name<<"\n";
std::cout<<phoneBook[i].age<<"\n";
std::cout<<phoneBook[i].phoneNumber<<"\n\n";
}
return 0;
}

Output:
Person 1:
Tutorial Series
6
555-555-0000
Person 2:
Frank Sinatra
15
555-555-1111
Person 3:
Bob Ross
30
555-555-2222
Node.h

/*
   Author | Date | Email
 Node: A self-referential member of a list. A self-referential object, such as a node, is an object that contains a reference member that refers to a class object of the same type. Nodes can be linked together to form useful data structures. Please note that this example uses pointers so please have an understanding of pointers before moving onto this example. Or move onto another section where pointers are not present. This header file will be important in later structures to come.
*/

typedef struct Node{
    Node* next;//the reference to the next node
    int data;//please note this can be any type of data
        //we are using an int for simplicity sake
}Node;
LinkedList.h

#include "Node.h"
#include <string>

@Author | Date | Email

LinkedList: Are collections of data members lined up in a row, insertions and deletions can be made anywhere in a linked list. Linked lists are linear collections of self-referential class objects, called nodes. A linked list is accessed by the first node in the list, each subsequent node is accessed through the link reference member in the previous node. Linked lists are dynamic, meaning they can grow or shrink in size as needed. Nodes of lists may not be held in continuous pieces of memory meaning they may be scattered in your memory.
Pros:
1) Inserting and deleting data does not require us to move/shift other elements which in turns slows down running time.

Cons:
1) Traversing the list can take a long time, to access a specific element
2) Requires more memory
3) Easily corrupted, can add data into middle easily

*/

class LinkedList{

public:
    Node* head;//a pointer to a Node

    /*
    Constructor: Initializes head to null, and creates a new Linked List.
    */
    LinkedList(){
        head=NULL;
    }

    /*
    Inserts an element "x" to the front of a LinkedList.
    */
    void insertAtFront(int x){
        //we must first create a place to store our reference and data
        Node *newNode;
        newNode = new Node;
        //assigning the data to given fields
        newNode -> data = x;
        newNode -> next=head;//setting the reference is the most
        //important part
        head = newNode;
    }

    /*
    Inserts an element "x" to the back of a LinkedList
    */
    void insertAtBack(int x){
        //just like before we need a new node to
        //link onto the list
        Node* newNode;
newNode = new Node;
newNode->data = x;
newNode->next = NULL;

//but now we need to find the end of the list
Node* endNode;
endNode = head;

if(endNode == NULL)
    endNode = newNode;
else{
    while(endNode->next != NULL)
        endNode = endNode->next;
    endNode->next = newNode; //when its all over we have the end of
                                //the list so we must link the two
}
}

/*
   Removes the first node from the LinkedList.
   You can modify this function to just delete
   rather than return the data in the deleted node.
   However, we will need this later.
*/
int removeFromFront(){
    int removedItem;
    removedItem = head->data; //store the data
    //now we need to reset the list
    if(head != NULL) {
        Node* toBeDeleted = head;
        removedItem = head->data;
        head = head->next;
        delete toBeDeleted; //free up the old space
    }
    return removedItem;
}

/*
   Removes the last node from LinkedList.
   You can modify this function to just delete
   rather than return the data in the deleted node. However, we will
   need this later.
*/
int removeFromBack()
{
    // we will approach this just like before however
    // we need to find the back of the list
    int removedItem;
    // if head is null there is no need to iterate
    if (head->next == NULL) {
        removedItem = head->data;
        delete head;
        head = NULL;
        return removedItem;
    }
    // if head has a link then we need to iterate through
    else {
        Node* nextToEnd = head;
        Node* end = head->next;
        while (end->next != NULL) {
            nextToEnd = end;
            end = end->next;
        }
        removedItem = end->data;
        delete end;
        nextToEnd->next = NULL;
    }
    return removedItem;
}

/*
    Linked lists also allow you to insert data members
    at any location in the LinkedList,
    InsertAt: Inserts element x at index position. x must be
    greater than zero and less than size of LinkedList.
*/
void insertAt(int x, int position){
    Node* newNode;
    newNode = new Node;
    newNode->data = x;
    newNode->next = NULL;
    if(head == NULL)
        head = newNode;
    else{
        Node* current;
current = head;
for(int i = 0; i < position-1; i++)
    current = current->next;
Node* temp = new Node;
temp = current->next;
current->next = newNode;
newNode->next = temp;
}
}

/*
Another useful function for a linked list is
a function to test whether or not an element
is present. Our function returns the index of
the found element. If no such element exists
a -1 is returned.
*/
int contains(int x){
    //if no nodes in our list
    if(head == NULL)
        return -1;
    Node* current = head;
    int index = 0;
    while(current->next != NULL){
        //make sure we don’t try to access NULL
        if(current->next == NULL)
            return -1;
        else if(current->data == x)
            return index;
        current = current->next; //move onto the next link
        index++;
    }
    //now we need to make sure to test
    //the last element in the list
    if(current->data == x)
        return index;

    return -1;
Returns the element at index.

int getByIndex(int index) {
    Node* current;
    current = head;
    for(int i = 0; i < index; i++)
        current = current->next;
    return current->data;
}

Time for you to write a function. This function
will remove an element x from index from the linked list if
x is present in the linked list.

Prints a representation of our list.

void print() {
    Node *current; // to keep track of the different nodes
    current = head; // set current to the start of the list
    while(current != NULL) {
        // now we will just iterate through
        std::cout<<"\t"<<current->data<<"\n";
        current = current->next;
    }
}

};
#include <iostream>
#include "LinkedList.h"

/*
 * LinkedListDemo: Exercises the LinkedList header file, in order
to test all the functions we just made. Remember that LinkedLists
are good at some things and not so good at others.
*/

int main(){
    //first create the LinkedList
    LinkedList list;
    //next populate the list
for(int i = 0; i<5; i++){
    list.insertAtFront(i*i);
}

//now print the list
std::cout<<"---Printing List---\n";
list.print();
//now test insertFromBack()
for(int i = 5; i<10; i++){
    list.insertAtBack(i*i);
}
std::cout<<"---Printing List---\n";
list.print();

//now test our remove functions
int a = list.removeFromFront();
printf("First Item should be removed\n");
std::cout<<"---Printing List---\n";
list.print();
int b = list.removeFromBack();
printf("Last Item should be removed\n");
std::cout<<"---Printing List---\n";
list.print();
//it was not necessary to store the values
//of our remove functions however to test we should
printf("Top of List removed: %d\n", a);
printf("Back of List removed: %d\n", b);
list.insertAt(999, 2);
std::cout<<"Inserting 999 at position 2:\n";
list.print();
list.insertAt(999, 7);
std::cout<<"Inserting 999 at position 7:\n";
list.print();
std::cout<<"list contains 1: "<<list.contains(1)<<"\n";
std::cout<<"list contains 999: "<<list.contains(999)<<"\n";
list.insertAt(999, 7);
std::cout<<"list contains 2: "<<list.contains(2)<<"\n";

}
---Printing List---
  16
  9
  4
  1
  0
  25
  36
  49
  64
  81

First Item should be removed
---Printing List---
  9
  4
  1
  0
  25
  36
  49
  64
  81

Last Item should be removed
---Printing List---
  9
  4
  1
  0
  25
  36
  49
  64

Top of List removed: 16
Back of List removed: 81
Inserting 999 at position 2:
  9
  4
  999
  1
  0
  25
  36
  49
#include “LinkedList.h”

/*
* Author | Email | Date
.Stack: A stack is a modified version of a linked list. Stacks use the last-in first-out (LIFO) policy. Meaning the last member to be inserted is the first member to be taken out. New nodes in stacks can only be added to the "top" (the front). Two functions used in a stack are "push" to add a member to the top, and "pop" to retrieve the last member inserted. We are going to take advantage of inheritance in order to derive most of our Data Structure Examples.

Pros:
1) Helps to manage data in a particular way (LIFO).
2) Variables are "destroyed" when you "pop" them off the stack, this gives you control over how memory is allocated and deallocated.
3) No need to free up unused space
4) Stacks are not easily corrupted, meaning you can only add to the top of a stack

Cons:
1) Stack memory is limited.
2) Too many objects on stack will increase chances of stack overflow.
3) You cannot randomly access a stack.
class Stack: public LinkedList{
public:

    Uses the parent class LinkedList's constructor in order to build a stack. Initializes head to NULL.
    */
    Stack(){
        LinkedList::LinkedList();
    }

    /*
    Returns the top of the stack.
    */
    int pop(){
        return LinkedList::removeFromFront();
    }

    /*
    Pushes an element x onto the top of the stack.
    */
    void push(int x){
        LinkedList::insertAtFront(x);
    }

    /*
    Uses LinkedList's print function to print out a representation of the stack.
    */
    void print(){
        std::cout<<"---Printing Stack---\n";
        LinkedList::print();
    }
};
#include <iostream>
#include "Stack.h"

/*
 StackDemo: Exercises the Stack header file. The import statement allows us to use the Stack class in any file we want. This demo is just a test to make sure everything is working as it should. Remember the Stack uses a LIFO policy, and utilizes functions such as pop, and push.
*/

int main(){
Stack stack;
// now let's populate it
for(int i = 0; i < 10; i++){
    stack.push(i*i*i);
    stack.print();
}
// now let's test the pop function
for(int i = 0; i < 10; i++){
    int x = stack.pop();
    std::cout<<"We just popped "<<x<<" off the stack.\n";
    std::cout<<"The stack now looks like: ";
    stack.print();
}
return 0;

Output:

---Printing Stack---
  0
---Printing Stack---
  1
    0
---Printing Stack---
  8
    1
    0
---Printing Stack---
  27
    8
    1
    0
---Printing Stack---
  64
    27
    8
    1
    0
---Printing Stack---
  125
    64
    27
    8
    1
---Printing Stack---
  729
  512
  343
  216
 125
  64
  27
  8
  1
  0
---Printing Stack---
  512
  343
  216
 125
  64
  27
  8
  1
  0
---Printing Stack---
  343
  216
 125
  64
  27
  8
  1
  0
---Printing Stack---
  216
 125
  64
  27
  8
  1
  0
---Printing Stack---
  0

We just popped 729 off the stack.
The stack now looks like: ---Printing Stack---
  512
  343
  216
We just popped 512 off the stack.
The stack now looks like: ---Printing Stack---

343
216
125
64
27
8
1
0

We just popped 343 off the stack.
The stack now looks like: ---Printing Stack---

216
125
64
27
8
1
0

We just popped 216 off the stack.
The stack now looks like: ---Printing Stack---

125
64
27
8
1
0

We just popped 125 off the stack.
The stack now looks like: ---Printing Stack---

64
27
8
1
0

We just popped 64 off the stack.
The stack now looks like: ---Printing Stack---

27
8
We just popped 27 off the stack. 
The stack now looks like: ---Printing Stack---
8
1
0

We just popped 8 off the stack. 
The stack now looks like: ---Printing Stack---
1
0

We just popped 1 off the stack. 
The stack now looks like: ---Printing Stack---
0

We just popped 0 off the stack. 
The stack now looks like: ---Printing Stack—

Queue.h

#include "LinkedList.h"

/*
 Author | Email | Date
 Queue: Uses the principle of first-in first-out (FIFO). Queues, once again, are linear collections of items. When items are inserted into the queue they are inserted into the tail of the queue, and when members are removed they are removed from the head of the queue. Insert definitions are often referred to as "enqueue" and remove functions are often referred to as "dequeue".

Pros:
1) Helps to manage memory in a particular way, (LIFO).
2) Not easily corrupted

Cons:
1) Random access is not possible.
*/
class Queue: public LinkedList{
public:

/*
 Uses LinkedList's constructor to initialize head to NULL.
*/
Queue(){
    LinkedList::LinkedList();
}

Returns the head of the queue.
*/
int dequeue(){
    return LinkedList::removeFromFront();
}

Inserts an element x on the tail of the queue.
*/
void enqueue(int x){
    LinkedList::insertAtBack(x);
}

Uses LinkedList's print function to print out a representation of our queue.
*/
void print(){
    std::cout<<"---Printing Queue---\n";
    LinkedList::print();
}
};

QueueDemo.cpp

#include <iostream>
#include "Queue.h"

QueueDemo: Exercises the Queue header file. The import statement allows us to use the Queue class in any file we want. This demo is just a demo and a test to make sure everything is working as it should. Remember the Queue uses a FIFO policy, and utilizes functions such as dequeue and enqueue.

int main(){

Queue queue;
//now let's populate it
for(int i = 0; i < 10; i++){
    queue.enqueue(i*i*i);
    queue.print();
}
//now let's test the pop function
for(int i = 0; i < 10; i++){
    int x = queue.dequeue();
    std::cout<<"We just dequeued "<<x" off the queue.\n";
    std::cout<<"The queue now looks like:\n";
    queue.print();
}
return 0;
8
---Printing Queue---
  0
  1
  8
  27
---Printing Queue---
  0
  1
  8
  27
  64
---Printing Queue---
  0
  1
  8
  27
  64
  125
  216
---Printing Queue---
  0
  1
  8
  27
  64
  125
  216
  343
---Printing Queue---
  0
  1
  8
  27
  64
  125
  216
  343
---Printing Queue---
  0
  1
  8
  27
  64
  125
  216
We just dequeued 0 off the queue.
The queue now looks like:

---Printing Queue---

0
1
8
27
64
125
216
343
512
729

We just dequeued 1 off the queue.
The queue now looks like:

---Printing Queue---

1
8
27
64
125
216
343
512
729

We just dequeued 8 off the queue.
The queue now looks like:

---Printing Queue---

8
27
64
125
216
343
512
729
We just dequeued 27 off the queue. 
The queue now looks like:
---Printing Queue---
  512
  729

We just dequeued 64 off the queue. 
The queue now looks like:
---Printing Queue---
  125
  216
  343
  512
  729

We just dequeued 125 off the queue. 
The queue now looks like:
---Printing Queue---
  216
  343
  512
  729

We just dequeued 216 off the queue. 
The queue now looks like:
---Printing Queue---
  343
  512
  729

We just dequeued 343 off the queue. 
The queue now looks like:
---Printing Queue---
  512
  729

We just dequeued 512 off the queue. 
The queue now looks like:
---Printing Queue---
  729

We just dequeued 729 off the queue. 
The queue now looks like:
---Printing Queue---
# Set.h

```cpp
#include "LinkedList.h"

/*
 Set: A set is a computer implementation of a
 mathematic finite set. Sets have no particular order,
 sets also contain no duplicate data members. What this
 means is: if we have a set with -1 present, if we later try
 to add a -1 a set will not perform this function.
 With sets you typically test whether a member is present
 rather that retrieving that member. This is why we made the
 contains function in the LinkedList.h file, to be implemented
 in our Set. Sets also define functions such as: union, intersection,
 difference, and subset. These functions are fun to make and great
 practice, why don’t you try making these.
 */

class Set: public LinkedList{
public:

    /*
     Constructor: utilizes LinkedLists constructor
     to initialize a Set. Set is initialized to NULL.
     */
    Set();
    LinkedList::LinkedList();
```
void insertAt(int x, int pos){
if(LinkedList::contains(x) == -1)
    LinkedList::insertAt(x, pos);
}

void add(int x){
if(LinkedList::contains(x) == -1)
    LinkedList::insertAtFront(x);
}

void print(){
    LinkedList::print();
}

void remove(int x){
    LinkedList::remove(x);
}

void removePoint(int pos){
    LinkedList::removePoint(pos);
}
/*
void remove(int position){
  if(position == -1)
    LinkedList::removeFromBack();
  else if (position == 0)
    LinkedList::removeFromFront();
}
*/

SetDemo.cpp

#include <iostream>
#include "Set.h"

int main(){
  Set set;
}
for(int i = 0; i < 5; i++){
    set.add(i*i*i);
}

set.print();

std::cout<<"Trying to add duplicate 64: \n";
set.add(64);
set.print();

std::cout<<"Removing from head\n";
set.remove(0);
set.print();

std::cout<<"Removing from tail\n";
set.remove(-1);
set.print();

set.insertAt(7777, 3);
std::cout<<"Inserting 7777 at index 3:\n";
set.print();

std::cout<<"Trying once more to add a duplicate member 7777:\n";
set.add(7777);
set.print();

return 0;
}
Output:

64
27
8
1
0

Trying to add duplicate 64:
64
27
8
1
0

Removing from head
27
8
1
0

Removing from tail
27
8
1
0

Inserting 7777 at index 3:
27
8
1
7777

Trying once more to add a duplicate member 7777:
27
8
1
7777
ArrayList.h

#include "LinkedList.h"

/*
ArrayList: Utilizes "LinkedList.h" to create a dynamic array. Dynamic arrays grow and shrink as needed and size is not required prior to creating the array. Array lists are great for accessing and initializing indices, and should include such functions. In our case our ArrayList can hold only ints, however ArrayList can be adapted to hold any type of data.
*/

class ArrayList: public LinkedList{
public:

    /*It is useful to keep track of the size*/
    long size;

    /*
    Constructor: Utilizes LinkedList's constructor to initialize head to NULL and size to 0.
    */
    ArrayList(){
        LinkedList::LinkedList();
        size = 0;
    }

    /*
    Returns the size of this ArrayList.
    */
    long getSize(){
        return size;
    }

    /*
Inserts element x at the beginning of the ArrayList.

```cpp
void add(int x){
    size++;
    LinkedList::insertAtFront(x);
}
```

InsertAt: Inserts element x at position. Position must be greater than zero, and less than size - 1.

```cpp
void insertAt(int x, int position){
    if(position < 0 || position > size - 1)
        return;
    size++;
    LinkedList::insertAt(x, position);
}
```

GetByIndex: Index must be greater than zero, and less that or equal to size - 1. Utilizes LinkedLists getByIndex function.

```cpp
int getByIndex(int index){
    if(index < 0 || index > getSize() - 1)
        return -1234;
    return LinkedList::getByIndex(index);
}
```
ArrayListDemo.cpp

#include <iostream>
#include "ArrayList.h"
/*
   Utilizes the ArrayList.h file in order to use and test ArrayList.h.
   We are going to test all the functions, and make sure
   everything is in working order.
*/
int main(){

    ArrayList arrayList;

    std::cout<<"Populating array:\n";
    for(int i = 0; i < 10; i++)
        arrayList.add(i*i*i);
    arrayList.print();
Output:

Populating array:
    729
    512
    343
    216
    125
    64
    27
Testing to see if 64 is present:
64 at index: 5
Testing getByIndex:
arrayList at 0: 729
arrayList[0]: 729
arrayList[1]: 512
arrayList[2]: 343
arrayList[3]: 216
arrayList[4]: 125
arrayList[5]: 64
arrayList[6]: 27
arrayList[7]: 8
arrayList[8]: 1
arrayList[9]: 0
Trying to access invalid index:
getByIndex returns: -1234
Inserting 89 at index 3:
    729
    512
    343
    89
    216
    125
    64
    27
    8
    1
    0

Map.h
Maps are not concerned with indices, instead they are concerned with unique identifiers called keys. When keys are accessed they return values. Maps associate keys with values. Our map maps an int to an int, however this class can be modified to map any data type to any other data type, though this mapping will serve our purpose right now.

class Map {
public:
  LinkedList keys;
  LinkedList values;

  Map(){
    keys = LinkedList();
    values = LinkedList();
  }

  void map(int key, int value){
    keys.insertAtFront(key);
    values.insertAtFront(value);
  }

  int getKey(int key){
    if(keys.head == NULL) return -1234;
    Node* tempKey;
    tempKey = keys.head;
    Node* tempVal;
    tempVal = values.head;
    while(tempKey != NULL) {
if(tempKey->data == key)
    return tempVal->data;
tempKey = tempKey->next;
tempVal = tempVal->next;
}
return -1234;
}

/*
getByValue: Accesses a key mapped to
value. If value is present. If no value is present a
faulty key is returned, this could cause problems
later on but serves our purposes right now.
*/
int getByValue(int value){
    if(values.head == NULL)
        return -1234;
    Node* tempKey;
tempKey = keys.head;
    Node* tempVal;
tempVal = values.head;
while(tempVal != NULL){
    if(tempVal->data == value)
        return tempKey->data;
    tempKey = tempKey->next;
    tempVal = tempVal->next;
}
return -1234;
}

/*
printKeys: Prints all the keys in our map.
Can you modify this to return all keys?
*/
void printKeys(){
    std::cout<<"Printing Keys: \n";
    keys.print();
}

/*
printValues: Prints all the values in our map.
Can you modify this to return all values?
*/
void printValues(){
    std::cout<<"Printing Values: \n";
    values.print();
Print: Prints a representation of our map.
Uses the format [key:value].
*/

void print(){
    Node* currKey = keys.head;
    Node* currVal = values.head;
    while(currKey != NULL){
        std::cout<<"["<<currKey->data;
        std::cout<<":"<<currVal->data<<"]\n";
        currKey = currKey->next;
        currVal = currVal->next;
    }
}

#include <iostream>
#include "Map.h"

int main(){

    Map map;
    map.map(0,123);
    map.map(56,1234);
    map.map(112,12345);
    map.print();

    std::cout<<"\n";
    std::cout<<"Testing getByKey: \n";
    std::cout<<"getByKey(0): "<<map.getByKey(0)<<"\n";
    std::cout<<"getByKey(56): "<<map.getByKey(56)<<"\n";
    std::cout<<"getByKey(112): "<<map.getByKey(112)<<"\n";
    std::cout<<"getByKey(1): "<<map.getByKey(1)<<"\n";

    std::cout<<"\n";
    std::cout<<"Testing getByValue: \n";
    std::cout<<"getByValue(123): "<<map.getByValue(123)<<"\n";
    std::cout<<"getByValue(1234): "<<map.getByValue(1234)<<"\n";
    std::cout<<"getByValue(12345): "<<map.getByValue(12345)<<"\n";
    std::cout<<"getByValue(-1): "<<map.getByValue(-1)<<"\n";
}
std::cout<<"\n";
std::cout<<"Testing printKeys: \n";
map.printKeys();

std::cout<<"\n";
std::cout<<"Testing printValues: \n";
map.printValues();
return 0;
}

Output:

[112:12345]
[56:1234]
[0:123]

Testing getByKey:
getByKey(0): 123
getByKey(56): 1234
getByKey(112): 12345
getByKey(1): -1234

Testing getByValue:
getByValue(123): 0
getByValue(1234): 56
getByValue(12345): 112
getByValue(-1): -1234
Testing printKeys:
Printing Keys:
  112
  56
  0

Testing printValues:
Printing Values:
  12345
  1234
  123

Node.py

Author | Date | Email
Node: A self-referential member of a list. A self-referential object, such as a node, is an object that contains a reference member that refers to a class object of the same type. Nodes can be linked together to form useful data structures.
class Node:
    """
    Constructor: Default constructor initializes
data and next to None.
    """
    def __init__(self, data=None, next=None):
        self.data = data
        self.next = next

    # you could also have more constructors
    # in here, along with setters and getters

    LinkedList.py
LinkedList: Are collections of data members lined up in a row, insertions and deletions can be made anywhere in a linked list. Linked lists are linear collections of self-referential class objects, called nodes. A linked list is accessed by the first node in the list, each subsequent node is accessed through the link reference member in the previous node. Linked lists are dynamic, meaning they can grow or shrink in size as needed. Nodes of lists may not be held in continuous pieces of memory meaning they may be scattered in your memory.

Pros:
1) Inserting and deleting data does not require us to move/shift other elements which in turns slows down running time.

Cons:
1) Traversing the list can take a long time, to access a specific element
2) Requires more memory
3) Easily corrupted, can add data into middle easily

```
class LinkedList:
    
    Constructor: Default constructor initializes head Node to None
    
    def __init__(self, head=None):
        self.head = head

    insertAtFront: Inserts data to the front of LinkedList
    
    def insertAtFront(self, data):
        # insert a check here to check if head is None
        # if head is none, init head to data
        newNode = Node()
        newNode.data = data
        newNode.next = self.head
        self.head = newNode
```
insertAtBack: Inserts data to the back of LinkedList.

```python
def insertAtBack(self, data):
    current = self.head
    if current:
        while current.next != None:
            current = current.next
        newNode = Node()
        newNode.data = data
        current.next = newNode
    else:
        self.head = Node()
        self.data = data
```

insertAtIndex: Inserts data at i into our linked list

```python
def insertAtIndex(self, data, i):
    newNode = Node()
    newNode.data = data
    if self.head is None:
        self.head = newNode
    return
    else:
        current = self.head
        for i in range(i - 1):
            current = current.next
        temp = Node()
        temp = current.next
        current.next = newNode
        newNode.next = temp
```

removeFromFront: Removes the head node and resets the head.

```python
def removeFromFront(self):
```
if self.head.next is None and self.head.data is None:
    print("No data")
    return

data = self.head.data
self.head = self.head.next
return data

removeFromBack: Removes the tail node

def removeFromBack(self):
    tail = self.head
    previous = tail
    while tail.next != None:
        previous = tail
        tail = tail.next
    previous.next = None
    return tail.data

contains: Test to see if testData is present in our Linked List

def contains(self, testData):
    if self.head is None:
        return False
    current = self.head
    if current.data == testData:
        return True
    else:
        while current.next is not None:
            if current.data == testData:
                return True
            current = current.next
    return False
Now you should add a function that deletes the node from position i. Then create a function that deletes the node with a specified element. Finally add one more function that reinitializes a certain nodes data.

printList: Prints a representation of our Linked List.

def printList(self):
    currentNode = self.head
    while(currentNode):
        print(str(currentNode.data))
        currentNode = currentNode.next

from LinkedList import LinkedList
import time

Exercises and tests the Linked List class.
Pt 1: We test are linked list.
Pt 2: We use the list data type Python has to offer.

Part One

list = LinkedList()
print("Testing insertAtFront")
for i in range(3):
    list.insertAtFront(i**3)
print("Printing Linked List:")
list.printList()
print("Timing how long it takes to populate")
print("our linked list with 1 million numbers")
list2 = LinkedList()
start = time.time()
for i in range(1000000):
    list2.insertAtFront(i)
end = time.time()
print("That took "+str(end-start)+" seconds")

print("Testing insertAtBack")
for i in range(3,6):
    list.insertAtBack(i**3)
print("Printing Linked List:")
list.printList()  

print("Testing removeFromBack")
list.removeFromBack()
print("Printing Linked List:")
list.printList() 

print("Testing removeFromFront")
list.removeFromFront()
print("Printing Linked List:")
list.printList() 

print("Testing contains")
print("list.contains(0): "+str(list.contains(0)))
print("list.contains(125): "+str(list.contains(125)))
print("Timing contains")
start = time.time()
list2.contains(0)
end = time.time()
print("That took "+str(end-start)+" seconds")
print("Testing insertAtIndex")
list.insertAtIndex(777,2)
print("Printing Linked List:")
list.printList()
list.insertAtIndex(222,4)
print("Printing Linked List:")
list.printList() 

","Part Two","pyList = []
print("Testing how long it takes to populate")
```python
print("a Python linked list")
start = time.time()
for i in range(1000000):
    pyList.append(i)
end = time.time()
print("That took "+str(end-start)+" seconds")

print("Testing Python contains")
start = time.time()
bool = 999999 in pyList
end = time.time()
print("That took "+str(end-start)+" seconds")

"""
The results: Our linked list is slower to populate, however our tests to see if an element is present are almost identical. All in all, you should utilize a language’s tools such as built in lists, but it is great knowledge and practice to be able to build your own.
"""
```

Output:
Testing insertAtFront
Printing Linked List:
8
1
Timing how long it takes to populate our linked list with 1 million numbers
That took 3.09680700302 seconds
Testing insertAtBack
Printing Linked List:
8
1
0
27
64
125
Testing removeFromBack
Printing Linked List:
8
1
0
27
64
Testing removeFromFront
Printing Linked List:
1
0
27
64
Testing contains
list.contains(0): True
list.contains(125): False
Timing contains
That took 0.196342945099 seconds
Testing insertAtIndex
Testing insertAtIndex
Printing Linked List:
1
0
777
27
64
Printing Linked List:
1
0
777
27
222
64
Testing how long it takes to populate a Python linked list
That took 0.162801980972 seconds
Testing Python contains
That took 0.018590927124 second

Stack.py

Author | Email | Date
Stack: A stack is a modified version of a linked list. Stacks use the last-in first-out (LIFO) policy. Meaning the last member to be inserted is the first member to be taken out. New nodes in stacks can only be added to the "top" (the front). Two functions used in a stack are "push" to add a member to the top, and "pop" to retrieve the last member inserted. We are going to take advantage of inheritance in order to derive most of our Data Structure Examples.

Pros:
1) Helps to manage data in a particular way (LIFO).
2) Variables are "destroyed" when you "pop" them off the stack, this gives you control how memory is allocated and deallocated.
3) No need to free up unused space
4) Stacks are not easily corrupted, meaning you can only add to the top of a stack

Cons:
1) Stack memory is limited.
2) Too many objects on stack will increase chances of stack overflow.
3) You cannot randomly access a stack.

```python
class Stack(LinkedList):
    """
    Constructor: Initializes head to None.
    """
    def __init__(self, head=None):
        self.head = head
    """
    push: Pushes data to the top of the Stack.
    """
    def push(self, data):
```
```
self.insertAtFront(data)

pop: Pops the top element off the top of the Stack.

def pop(self):
    return self.removeFromFront()

printStack: Prints a representation of our Stack.

def printStack(self):
    print("Printing Stack")
    self.printList()

StackTest.py

from Stack import Stack
import time

Author | Email | Date
Exercises and tests our stack class.
Part One: We test our stack
Part Two: We test the stack python has to offer

Part One

stack = Stack()
print("Testing push")
stack.push(5)
stack.push(4)
stack.push(3)
stack.push(2)
stack.push(1)
stack.push(0)
stack.printStack()

print("Testing pop")
print("We just popped: "+str(stack.pop()))
print("Stack now looks like")
stack.printStack()
print("We just popped: "+str(stack.pop()))
print("Stack now looks like")
```
stack.printStack()

stack2 = Stack()
print("Populating Stack with 1 Million Numbers")
start = time.time()
for i in range(1000000):
    stack.push(i)
end = time.time()
print("That took "+str(end-start)+" seconds")

"""
Part Two: Using a list from Python.
The standard data type List in python comes pre equipped with pop and append
A list in Python is mutable, meaning you can change it. List aos comes with nice functions like sort and del. You can index individual elemets in a list much like an array.
Go online to check out more awesome functions list has to offer.
"""

stack3 = []
print("Populating Python TStack with 1 Million Numbers")
start = time.time()
for i in range(1000000):
    stack3.append(i)
end = time.time()
print("That took "+str(end-start)+" seconds")

stack4 = [4,5,7,1,2]
print("stack4 pre sorted: "+str(stack4))
stack4.sort()
print("stack4 post sorted: "+str(stack4))
stack4.insert(0, "New Element")
print("'New Element should be at index 0'")
print(stack4)
print("Testing del stack4[0]")
del stack4[0]
print(stack4)
print("Testing stack4.remove(7)")
stack4.remove(7)
print(stack4)
print("Printing stack4[1:3]")
print(stack4[1:3])

Output:
Testing push
Printing Stack
0
1
2
3
4
4
5
Testing pop
We just popped: 0
Stack now looks like
Printing Stack
1
2
3
4
5
We just popped: 1
Stack now looks like
Printing Stack
2
3
4
5
Populating Stack with 1 Million Numbers
That took 3.49722695351 seconds
Populating Python Stack with 1 Million Numbers
That took 0.167644023895 seconds
stack4 pre sorted: [4, 5, 7, 1, 2]
stack4 post sorted: [1, 2, 4, 5, 7]
'New Element should be at index 0'
['New Element', 1, 2, 4, 5, 7]
Testing del stack4[0]
[1, 2, 4, 5, 7]
Testing stack4.remove(7)
[1, 2, 4, 5]
Printing stack4[1:3]
[2, 4]
from LinkedList import LinkedList

Author | Email | Date
Queue: Uses the principle of first-in first-out (FIFO).
Queues, once again, are linear collections of items. When items are inserted into the queue they are inserted into the tail of the queue, and when members are removed they are removed from the head of the queue. Insert definitions are often referred to as "enqueue" and remove functions are often referred to as "dequeue".

Pros:
1) Helps to manage memory in a particular way, (LIFO).
2) Not easily corrupted

Cons:
1) Random access is not possible.

class Queue(LinkedList):

Constructor: Initializes head to None.

def __init__(self, head=None):
    self.head = head

tenqueue: Inserts data to the head of Queue

def enqueue(self, data):
    self.insertAtBack(data)

dehqueue: Returns the bottom element of the queue.

def dequeue(self):
    return self.removeFromFront()

printQueue: Prints a representation of our Queue.
def printQueue(self):
    print("Printing Queue")
    self.printList()

QueueTest.py

from Queue import Queue#our queue

""
Exercises and tests our user made queue.
Python also offers a queue, priority queue,
and LIFO queue (a stack). We will not be testing
those right now, but it would follow the
pattern of Linked List and Stack. You should
go ahead a try.
""

queue = Queue()
print("Testing Enqueue")
for i in range(10):
    queue.enqueue(i)
queue.printQueue()

print("Testing Dequeue")
for i in range(10):
    print("Dequeue round "+str(i))
    print("dequeue: “+str(queue.dequeue()))
Output:

Testing Enqueue
Printing Queue
0
1
2
3
4
5
6
7
8
9
Testing Dequeue
Dequeue round 0
dequeue: 0
Dequeue round 1
dequeue: 1
Dequeue round 2
dequeue: 2
Dequeue round 3
dequeue: 3
Dequeue round 4
dequeue: 4
Dequeue round 5
dequeue: 5
Dequeue round 6
dequeue: 6
UserSet.py

from LinkedList import LinkedList

Author | Email | Date

Set: A set is a computer implementation of a mathematic finite set. Sets have no particular order, sets also contain no duplicate data members. What this means is: if we have a set with -1 present, if we later try to add a -1 a set will not perform this function. With sets you typically test whether a member is present rather that retrieving that member. This is why we made the contains function in the LinkedList.py file, to be implemented in our Set. Sets also define functions such as: union, intersection, difference, and subset. These functions are fun to make and great practice, why don’t you try making these.

class UserSet(LinkedList):
Constructor: Creates a Set, initializes head to None

```python
def __init__(self, head=None):
    self.head = head
```

Insert: Inserts data at index i. Checks to make sure data is not already present in set, as Sets contain no duplicate values.

```python
def insert(self, data, i):
    if not self.contains(data):
        self.insertAtIndex(data, i)
```

Append: Inserts data to the front of our Set. Checks to ensure data is not present in our Set

```python
def append(self, data):
    if not self.contains(data):
        self.insertAtFront(data)
```

Remove: If i = -1 the tail of our set is removed. If i = 0 the head of our set is removed.

```python
def remove(self, i):
    if i == -1:
        self.removeFromBack()
    elif i == 0:
        self.removeFromFront()
```
printSet: Prints a representation of our set.

```python
def printSet(self):
    print("Printing Set")
    self.printList()
```

Now is the time, if you made a delete function that deletes specified elements, to utilize that function in order to delete specific members.

```
SetTest.py
```

from UserSet import UserSet # Our Set
Exercises and tests our user set vs. Python's pre made set.
Part One: We test our Set
Part Two: We test Python's Set

Part One

userSet = UserSet()
print("Populating Set")
for i in range(5):
    userSet.append(i**3)
userSet.printSet()
print("Trying to add duplicate value 64")
userSet.append(64)
userSet.printSet()

Part Two

pySet = []
print("Populating pySet")
for i in range(5):
    pySet.append(i**3)
print(pySet)
print("Trying to add duplicate value 64")
userSet.append(64)
print(pySet)

Sets are a data type readily available after Python 2.6
just declare a set how we declared pySet. Sets can be either
mutable or immutable. Sets also come with many great functions
to compare two sets, combine sets, or check sets. Research Sets some
more online. They are a great tool.
Output:

Populating Set
Printing Set
 64
 27
  8
   1
    0
Trying to add duplicate value 64
Printing Set
 64
 27
  8
   1
    0
Populating pySet
[0, 1, 8, 27, 64]
Trying to add duplicate value 64
[0, 1, 8, 27, 64]
from sets import Set


Author | Email | Date
A tuple is like a list except it is immutable, meaning once it has been initialized, you can not add to nor change the tuple.

#tuples are created with parenthesis
#and can hold different data types
grades = ('Mike',45,'Lisa', 99)
print('grades: ' + str(grades))

#you can access tuples how you access lists
print('grades[1]: ' + str(grades[1]))

#you can check to see if an item is in a list
print('45 in grades: ' + str((45 in grades)))

# you can also iterate through a tuple just like a list
for grade in grades:
    print('grade: ' + str(grade))

Built in tuple functions include:
cmp(t1, t2), len(), max(), min(), tuple()
Map.py

from LinkedList import LinkedList

"""
Author | Email | Date
Maps are not concerned with individual indexes, instead they are concerned with unique
identifiers called “keys”. When keys are accessed they return values. Maps associate keys with
values. Our map can map any data type to another data type.
"""

class Map:

"""
Constructor: Initializes our map using
two LinkedLists.
"""
def __init__(self):
    self.keys = LinkedList()
    self.values = LinkedList()

"""
map: Maps key to value
"""
def map(self, key, value):
    self.keys.insertAtFront(key)
    self.values.insertAtFront(value)
getByKey: Returns the value mapped to key

def getKey(self, key):
    if self.keys.head is None:
        print("No Value Present")
        return
    currKey = self.keys.head
    currVal = self.values.head
    while currKey is not None:
        if currKey.data is key:
            return currVal.data
        currKey = currKey.next
        currVal = currVal.next

getByValue: Returns the key mapped to value

def getByValue(self, value):
    if self.values.head is None:
        print("No Value Present")
        return
    currKey = self.keys.head
    currVal = self.values.head
    while currVal is not None:
        if currVal.data is value:
            return currKey.data
        currKey = currKey.next
        currVal = currVal.next

printMap: Prints a representation of our Map.
def printMap(self):
    currKey = self.keys.head
    currVal = self.values.head
    while(currKey is not None):
        print("["+str(currKey.data)+":"+str(currVal.data)+"]")
        currKey = currKey.next
        currVal = currVal.next

DictDemo.py

from Map import Map

Exercises and tests our Map class along side the default "Dictionary" type Python has to offer.
Part One: We test our Map class
Part Two: We test Python's dictionary

....
Part One

```python
map = Map()
map.map(1,"One")
map.map(2,"Two")
map.map(3,"Three")
map.map(4,"Four")
map.map(5,"Five")
map.printMap()

print("Testing getByKey")
for i in range(1,6):
    print("map.getByKey("+str(i)+"):"+ str(map.getByKey(i)))

print("Testing getValue")
print('map.getValue("One"):')
map.getValue("One")
print('map.getValue("Two"):')
map.getValue("Two")
```

```
Part Two

```python

zipcodes = {
    "Des Moines": 50301,
    "Tallahassee": 32301,
    "Honolulu": 96801
}

# we are using a key to access a value
print('zipcodes["Honolulu"]: ' + str(zipcodes['Honolulu']))

# you can add new entries or update
print("zipcodes before: "+str(zipcodes))
zipcodes['Des Moines'] = 10000
zipcodes['Sacramento'] = 45678
print("zipcodes after: "+str(zipcodes))

# what about a list as a value, or a dict as a value?
# people, have a phone number and an age...
people = {'Bella': [5559089999, 23], 'Mike': [5557890000, 55]}
print("people['Bella']: "+ str(people['Bella']))
print("people['Bella'][0]: "+ str(people['Bella'][0]))

# you can iterate through a list much
# the same way we have learned before
for key in zipcodes:
    print('Key: '+key+' Value: '+str(zipcodes[key]))

There are many functions to deal with dictionaries:
cmp, len, str, type, clear, copy, fromkeys, get, has_key,
items, keys, setdefault, update, values

print("zipcodes.items(): "+str(zipcodes.items()))
print("zipcodes.values(): "+str(zipcodes.values()))
```
Output:

[5:Five]
[4:Four]
[3:Three]
[2:Two]
[1:One]

Testing getByKey
map.getByKey(1): One
map.getByKey(2): Two
map.getByKey(3): Three
map.getByKey(4): Four
map.getByKey(5): Five

Testing getByValue
map.getByValue("One"):
map.getByValue("Two"):

zipcodes["Honolulu"]: 96801
zipcodes before: {'Tallahassee': 32301, 'Honolulu': 96801, 'Des Moines': 50301}
zipcodes after: {'Sacramento': 45678, 'Tallahassee': 32301, 'Honolulu': 96801, 'Des Moines': 10000}

people['Bella']: [5559089999, 23]
people['Bella'][0]: 5559089999

Key: Sacramento Value: 45678
Key: Tallahassee Value: 32301
Key: Honolulu Value: 96801
Key: Des Moines Value: 10000

zipcodes.items(): [('Sacramento', 45678), ('Tallahassee', 32301), ('Honolulu', 96801), ('Des Moines', 10000)]
zipcodes.values(): [45678, 32301, 96801, 10000]
```python
from collections import defaultdict

"""
A defaultdict is a dict where the value is a specified data type. In our case we are going to have a dictionary be that specific data type. This is useful for representing graphs. A graph data structure has edges and weights. For example, we are going to graph cities and the weights will be the distance between those cities.
"""

cities = defaultdict(dict)
#our default value is a dict

#say we want to graph distances from L.A. to various other cities
print("Populating graph")
for dest, dist in ("Portland", 963.4), ("Sacramento", 361):
    cities["Los Angeles"] = dist
print(cities)

#now we have the distances from L.A. to two different cities, we can make the tuples
#three elements long in order to
#for our for loop to be even more general
#or you could use this technique to read from
#a file and start mapping yourself
print('Printing cities["Los Angeles"]')
print(cities["Los Angeles"])```

GraphDemo.py

from collections import defaultdict

"""
A defaultdict is a dict where the value is a specified data type. In our case we are going to have a dictionary be that specific data type. This is useful for representing graphs. A graph data structure has edges and weights. For example, we are going to graph cities and the weights will be the distance between those cities.
"""

cities = defaultdict(dict)
#our default value is a dict

#say we want to graph distances from L.A. to various other cities
print("Populating graph")
for dest, dist in ("Portland", 963.4), ("Sacramento", 361):
    cities["Los Angeles"] = dist
print(cities)

#now we have the distances from L.A. to two different cities, we can make the tuples
#three elements long in order to
#for our for loop to be even more general
#or you could use this technique to read from
#a file and start mapping yourself
print('Printing cities["Los Angeles"]')
print(cities["Los Angeles"])
print('Printing cities["Los Angeles"]["Portland"]')
print(cities["Los Angeles"]["Portland"])

Output:

Populating graph
defaultdict(<type 'dict'>, {'Los Angeles': {'Sacramento': 361, 'Portland': 963.4}})
Printing cities["Los Angeles"]
{'Sacramento': 361, 'Portland': 963.4}
Printing cities["Los Angeles"]["Portland"]
963.4

ArrayDemo.java

/*
 * Author | Email | Date
 * ArrayDemo: Outlines different techniques you can use when dealing with arrays within Java.
 */
public class ArrayDemo{
    public static void main(String[] args){
        int[] numbers = new int[10];
        for(int i = 0; i < 10; i++){
            numbers[i] = (int)(Math.random()*567);
            System.out.printf("numbers[%d]: %d\n",i, numbers[i]);
        }
        //in order to add to array, we can use individual indexes
        numbers[0]=100;//was 0
        System.out.printf("numbers[0]: %d\n", numbers[0]);

        //in order to resize array we must create a new array
        int[] numbersTwo = new int[15];
for(int i = 0; i<15; i++)
{
    numbersTwo[i] = (i<10) ? numbers[i] : i*2;
    System.out.printf("numbersTwo[%d]: %d\n", i, numbersTwo[i]);
}

/*
Arrays are "easy" to implement, access, and delete members. As such, they are the most widely used data structure. However, if you want to insert an element at an already given position, you must create a new array one size bigger, shift everything, and initialize the given index. This is a time consuming task, especially if the array is of large size. The same is true about deleting a member.
*/

//we want to add 7 to "numbersTwo", at index 3, without altering the rest of the members
int[] arrayThree = new int[16]; //the array to transfer into
for(int i = 0; i < 16; i++)
{
    if(i == 3)
        arrayThree[i] = 7;
    else
        arrayThree[i] = (i<3) ? numbersTwo[i] : numbersTwo[i-1];
    System.out.printf("arrayThree[%d]: %d\n", i, arrayThree[i]);
}

/*
Searching for a member in an unsorted array, can at worst take O(n) running time to finish, where n is the size of the array. However if array is sorted, you can use logic to find members more quickly. In this case O(logN) is the worst case running time.
*/

//we'll initialize index to -1 so if this returns we know no element exists
//since our array is unsorted we will brute force
int index = -1;
int target = 100; //this can be any number
//we want to check for
for(int i = 0; i < numbersTwo.length; i++)
    if(numbersTwo[i] == target)
index = i;
System.out.printf("Index where target was found: %d\n",index);
}

Output:

numbers[0]: 504
numbers[1]: 385
numbers[2]: 273
numbers[3]: 481
numbers[4]: 384
numbers[5]: 195
numbers[6]: 208
numbers[7]: 219
numbers[8]: 77
numbers[9]: 23
numbers[0]: 100
numbersTwo[0]: 100
numbersTwo[1]: 385
numbersTwo[2]: 273
numbersTwo[3]: 481
numbersTwo[4]: 384
numbersTwo[5]: 195
numbersTwo[6]: 208
numbersTwo[7]: 219
numbersTwo[8]: 77
numbersTwo[9]: 23
numbersTwo[10]: 20
numbersTwo[11]: 22
numbersTwo[12]: 24
numbersTwo[13]: 26
numbersTwo[14]: 28
arrayThree[0]: 100
arrayThree[1]: 385
arrayThree[2]: 273
arrayThree[3]: 7
arrayThree[4]: 481
arrayThree[5]: 384
arrayThree[6]: 195
arrayThree[7]: 208
arrayThree[8]: 219
arrayThree[9]: 77
arrayThree[10]: 23
arrayThree[11]: 20
arrayThree[12]: 22
arrayThree[13]: 24
arrayThree[14]: 26
arrayThree[15]: 28
Index where target was found: 0
Node.java

/*
   @author | Date | Email
Node: A self-referential member of a list. A self-referential object, such as a node, is an object that contains a reference member that refers to a class object of the same type. Nodes can be linked together to form useful data structures.
*/

class Node {

    Object data; // can hold one of any Object
    Node next; // self-referential member, refers to the next node

    /*
       Constructor: Creates a Node that refers to Object o.
    */
    Node(Object o){
        this(o, null);
    }

    /*
       Constructor: Creates a Node that refers to Object o and to the next Node in the LinkedList.
    */
    Node(Object o, Node nextNode){
        data = o; // this node refers to Object o
        next = nextNode; // set next to refer to next
    }

    /*
       Returns a reference to the Object in this node
    */
    Object getObject(){
        return data;
    }

    /*
       Returns the next node
    */
LinkedList.java

/*@Author | Date | Email
LinkedList: A collection of data members lined up in a row, insertions and deletions can be made anywhere in a linked list. Linked lists are linear collections of self-referential class objects, called nodes. A linked list is accessed by the first node in the list, each subsequent node is accessed through the link reference member in the previous node. Linked lists are dynamic, meaning they can grow or shrink in size as needed. Nodes of lists may not be held in continuous pieces of memory meaning they may be scattered in your memory.

Pros:
1)Inserting and deleting data does not require us to move/shift other elements which in turns slows down running time.

Cons:
1)Traversing the list can take a long time, to access a specific element
2)Requires more memory
3)Easily corrupted, can add data into middle easily
*/

public class LinkedList {
   Node firstNode;
   Node lastNode;
   String name;  //just an identifier for different lists

   /*
   Constructor: Constructs an empty list
*/
with 'nameInput' as the name.
 */
public LinkedList( String nameInput ) {
    name = nameInput;
    firstNode = lastNode = null;
}

/*
   Constructor: Construct an empty List with
   "list" as the name
 */
public LinkedList(){
    this( "list" );
}

/*
   Insert an Object at the front of the List
   If List is empty, firstNode and lastNode will refer to
   the same object. Otherwise, firstNode refers to new node.
 */
public void insertAtFront( Object insertItem )
{
    if ( isEmpty() )
        firstNode = lastNode = new Node(insertItem);
    else
        firstNode = new Node(insertItem, firstNode);
}

/*
   Insert an Object at the end of the List
   If List is empty, firstNode and lastNode will refer to
   the same object. Otherwise, lastNode's next instance
   variable refers to new node.
 */
public void insertAtBack( Object insertItem )
{
    if ( isEmpty() )
        firstNode = lastNode = new Node( insertItem );
else
    lastNode = lastNode.next = new Node( insertItem );
}

/*
Removes the first node from the List.
*/
public Object removeFromFront(){
    Object removeItem = null;
    // can you add an exception to test if
    // list is empty?
    removeItem = firstNode.data;  // retrieve the data
    // reset the firstNode and lastNode references
    if (firstNode.equals(lastNode))
        firstNode = lastNode = null;
    else
        firstNode = firstNode.next;
    return removeItem;
}

/*
Removes the last node from the List.
*/
public Object removeFromBack(){
    Object removeItem = null;
    // can you add an exception to test if
    // list is empty?
    removeItem = lastNode.data;
    // reset the firstNode and lastNode references
    if (firstNode.equals(lastNode))
        firstNode = lastNode = null;
    else {
        Node current = firstNode;
while (current.next != lastNode) // not last node
    current = current.next; // move to next node
lastNode = current;
current.next = null;
}
return removeItem;
}

// Return true if the List is empty
public boolean isEmpty()
{
    return firstNode == null;
}

// Output the List contents
public void print()
{
    if (isEmpty()) {
        System.out.println("Empty "+name);
        return;
    }
    System.out.println("The "+name +" is:");
    Node current = firstNode;
    while (current != null) {
        System.out.print("-->"+current.data.toString() +"\n");
        current = current.next;
    }
    System.out.println("\n");
}
}
Exercises and tests our linked list class.

```java
public class LinkedListDemo {
    public static void main(String[] args) {
        LinkedList list = new LinkedList("First List");
        list.insertAtFront('a');
        list.insertAtFront('b');
        list.print();
        list.insertAtBack('c');
        list.insertAtBack('d');
        list.print();
        LinkedList list2 = new LinkedList("Second List");
        list2.insertAtBack(1);
        list2.insertAtBack(2);
        list2.insertAtBack(3);
        list2.insertAtBack(4);
        list2.insertAtFront(5);
        list2.print();
    }
}
```

Output:

The First List is:
```
-->b
-->a
```

The First List is:
```
-->b
-->a
-->c
-->d
```

The Second List is:
```
-->5
-->1
-->2
```
Stack.java

/*
Author | Email | Date
Stack: A stack is a modified version of a linked list. Stacks use the last-in first-out (LIFO) policy. Meaning the last member to be inserted is the first member to be taken out. New nodes in stacks can only be added to the "top" (the front). Two functions used in a stack are "push" to add a member to the top, and "pop" to retrieve the last member inserted.

Pros:
1) Helps to manage data in a particular way (LIFO).
2) Variables are "destroyed" when you "pop" them off the stack, this gives you control how memory is allocated and deallocated.
3) No need to free up unused space
4) Stacks are not easily corrupted, meaning you can only add to the top of a stack

Cons:
1) Stack memory is limited.
2) Too many objects on stack will increase chances of stack overflow.
3) You cannot randomly access a stack.
*/

public class Stack extends LinkedList {

    /*
    Constructor: Constructs an empty stack with 'stack' as the name.
    */
    public Stack(){
        super( "stack" );
    }

    /*
    Pushes an Object o onto the top of the stack.
    */
public void push(Object o) {
    insertAtFront(o);
}

/**
   * Returns the top node of the stack.
   */
public Object pop() {
    return removeFromFront();
}

/**
   * Returns true if stack is empty
   */
public boolean isEmpty() {
    return super.isEmpty();
}

/**
   * Displays the stack onto the screen.
   */
public void print() {
    super.print();
}

//can you add a peek option?
//peek displays the top element without removing it
Exercises and tests our Stack class.

public class StackDemo{
    public static void main(String[] args){
        Stack stack = new Stack();
        System.out.println("Testing Push");
        stack.push("hello");
        for(int i = 0; i < 12; i++){
            stack.push((int)(Math.random()*300));
        }
        stack.print();
        System.out.println("Testing Pop");
        for(int i = 0; i < 12; i++){
            System.out.printf("Pop #%d: ",i);
            System.out.println(stack.pop());
        }
        stack.print();
    }
}
/*
    Java also offers a pre made Stack.
    You should research and play around with this
    and our user created Stack.
*/

Output:

Testing Push
The stack is:
    -->117
    -->41
    -->37
    -->3
    -->299
    -->221
    -->19
    -->212
    -->297
    -->51
    -->243
    -->190
Testing Pop
Pop #0: 117
Pop #1: 41
Pop #2: 37
Pop #3: 3
Pop #4: 299
Pop #5: 221
Pop #6: 19
Pop #7: 212
Pop #8: 297
Pop #9: 51
Pop #10: 243
Pop #11: 190
The stack is:
   -->hello

Queue.java

/*
   Author | Email | Date
Queue: uses the principle of first-in first-out (FIFO).
Queues, once again, are linear collections of items. When items are inserted into the queue they are inserted into the tail of the queue, and when members are removed they are removed from the head of the queue. Insert definitions are often referred to as "enqueue" and remove functions are often referred to as "dequeue".

Pros:
1) Helps to manage memory in a particular way, (LIFO).
2) Not easily corrupted

Cons:
1) Random access is not possible.

```
public class Queue extends LinkedList {

    /*
     * Constructor: constructs an empty queue with "queue" as name.
     */
    public Queue() {
        super( "queue" );
    }

    /*
     * Inserts an Object o to the tail of the queue
     */
    public void enqueue(Object o){
        insertAtBack( o );
    }

    /*
     * Removes an object o from the head of the queue.
     */
    public Object dequeue(){
        return removeFromFront();
    }

    /*
     * Returns true if queue is empty
     */
    public boolean isEmpty() {
        return super.isEmpty();
    }
}
```
/*
 * Prints out a representation of the queue.
 */
public void print() {
    super.print();
}
public class QueueDemo{
    public static void main(String[] args){
        Queue queue = new Queue();
        System.out.println("Testing enqueue");
        for(int i = 0; i < 10; i++)
            queue.enqueue(i*i*i);
        queue.print();

        System.out.println("Testing dequeue");
        for(int i = 0; i < 10; i++){
            System.out.printf("dequeue #%d: ", i);
            System.out.println(queue.dequeue());
        }
        queue.print();
    }
}

Output:

Testing enqueue
The queue is:
    -->0
    -->1
    -->8
    -->27
    -->64
    -->125
Testing dequeue
dequeue #0: 0
dequeue #1: 1
dequeue #2: 8
dequeue #3: 27
dequeue #4: 64
dequeue #5: 125
dequeue #6: 216
dequeue #7: 343
dequeue #8: 512
dequeue #9: 729
Empty queue

UserSet.java

/**<p>Set: A set is a computer implementation of a mathematic finite set. Sets have no particular order, sets also contain no duplicate data members. What this means is: if we have a set with -1 present and we later try to add a -1 a set will not perform this function. With sets you typically test whether a member is present rather that retrieving that member. Sets also define functions such as: union, intersection,difference, and subset. These functions are fun to make and great practice, why don’t you try making these. This is a barebones representation of a Set, however it is a good start.</p>*/

public class UserSet extends LinkedList{

 /**<p>Constructor: Default constructor initializes our Set, with head == NULL</p>*/

public UserSet(String name){
    super(name);
public void append(Object obj){
    if(this.contains(obj) == 0)
        insertAtFront(obj);
}

public int contains(Object obj){
    Node current = firstNode;
    while(current != null){
        if(current.data.equals(obj))
            return 1;
        current = current.next;
    }
    return 0;
}

public class UserSetTest{
    public static void main(String[] args){
        UserSet set = new UserSet("Set");
        System.out.println("Populating Set");
        for(int i = 0; i < 10; i++)
            set.append(i*i*i);
        set.print();
        System.out.println("Trying to add duplicate member 64");
        set.append(64);
        set.print();
    }
}
Output:

Populating Set
The Set is:
  -->729
  -->512
  -->343
  -->216
  -->125
  -->64
  -->27
  -->8
  -->1
  -->0

Trying to add duplicate member 64
The Set is:
  -->729
  -->512
  -->343
  -->216
  -->125
  -->64
  -->27
  -->8
  -->1
  -->0

UserArrayList.java

/*
   Author | Email | Date

An ArrayList is like an array however it is dynamic.
And because we are using "Object" it can hold different
data types. Java does offer a pre made Array List so I would recommend researching that. We must be able to add and delete members at certain indices. Duplicate members are ok.

*/

public class UserArrayList extends LinkedList{

    /*
     */
    public UserArrayList(String name){
        super(name);
    }

    /*
     append: Inserts obj at the front of the Array List
     */
    public void append(Object obj){
        insertAtFront(obj);
    }

    /*
     insert: Inserts obj at index i
     */
    public void insert(Object obj, int i){
        Node newNode = new Node(obj);
        Node current = firstNode;
        Node previous = current;
        for(int j = 0; j < i; j++){
            previous = current;
            current = current.next;
        }
        previous.next = newNode;
        newNode.next = current;
    }
/*
 * remove: Removes the object at index j.
 */

public void remove(int j){
    Node current = firstNode;
    Node previous = current;
    Node next = current;
    for(int i = 0; i < j; i++){
        previous = current;
        current = current.next;
        next = current.next;
    }
    previous.next = next;
}

/*
 * We are also missing a function to reinitialize an index
 * with a new Object, why don’t you try creating that.
 */
ArrayListTest.java

/*
   Author | Email | Date
   Exercises and tests our UserArrayList class.
   Java does offer a pre made array list so you
   should research that and have some fun with it.
*/

public class ArrayListTest{

    public static void main(String[] args){
        UserArrayList arraylist = new UserArrayList("Array List");
        System.out.println("Populating Array List");
        for(int i = 0; i < 5; i++)
            arraylist.append(i*i*i);
        arrayList.print();
        System.out.println("Inserting 77 at index 2");
        arrayList.insert(77,2);
        arrayList.print();
        System.out.println("Inserting 777 at index 4");
        arrayList.insert(777,4);
        arrayList.print();
        System.out.println("Removing Object at index 5");
        arrayList.remove(5);
        arrayList.print();
        System.out.println("Removing Object at index 3");
        arrayList.remove(3);
        arrayList.print();
    }
}
Output:

Populating Array List
The Array List is:
  -->64
  -->27
  -->8
  -->1
  -->0

Inserting 77 at index 2
The Array List is:
  -->64
  -->27
  -->77
  -->8
  -->1
  -->0

Inserting 777 at index 4
The Array List is:
  -->64
  -->27
  -->77
  -->8
  -->777
  -->1
  -->0
Removing Object at index 5
The Array List is:
   -->64
   -->27
   -->77
   -->8
   -->777
   -->0

Removing Object at index 3
The Array List is:
   -->64
   -->27
   -->77
   -->777
   -->0

Dictionary.java

/*
   Maps (a.k.a. dictionaries) are not concerned with indexes, instead they are concerned with unique identifiers called keys. When keys are accessed they return values. Maps associate keys with values. In the case of our java dictionary, we are mapping any Object to any other Object.
*/

public class Dictionary extends LinkedList{

   LinkedList key;
   LinkedList value;

   /*
      Constructor: constructs a Dictionary with two Linked Lists, Keys and Values.
   */
   public Dictionary(){

key = new LinkedList("Keys");
value = new LinkedList("Values");

/**
 * Returns true if Object k is present in key.
 */
public boolean hasKey(Object k)
{
    Node current = key.firstNode;
    while (current != null)
    {
        if (current.data.equals(k))
            return true;
    }
    return false;
}

/**
 * Maps k to v
 */
public void map(Object k, Object v)
{
    key.insertAtFront(k);
    value.insertAtFront(v);
}

/**
 * A private function not to be used outside of get(Object k)
 * Returns the index of Object k
 */
private int getIndex(Object k)
{
    int foundIndex = 0;
    Node current = key.firstNode;
    while (current != null) {
        foundIndex++;
        if (current.data.equals(k))
            return foundIndex;
        current = current.next;
    }
    return foundIndex;
}

/**
 * Returns the value of the Object k, if Object k is present in dictionary.
 */
public Object get(Object k){
    int index = getIndex(k);
    int i = 0;
    Object v = null;
    Node current = value.firstNode;
    while(current != null){
        i++;
        if(i == index)
            return current.data;
        current = current.next;
    }
    return v;
}

/*
 *  Prints a representation of the dictionary object.
 */
public void print(){
    Node keyCurrent = key.firstNode;
    Node valueCurrent = value.firstNode;
    while(keyCurrent != null){
        System.out.println("["+keyCurrent.data+" : "+
            valueCurrent.data+"]");
        keyCurrent = keyCurrent.next;
        valueCurrent = valueCurrent.next;
    }
}

}
public class DictionaryTest{
    public static void main(String[] args){
        Dictionary dict = new Dictionary();
        System.out.println("Mapping Dictionary");
        dict.map(1,"One");
        dict.map(2,"Two");
        dict.map(3,"Three");
        dict.print();
        System.out.println("Testing get");
        for(int i = 1; i<4;i++)
            System.out.printf("dict[%d]: %s\n",i,dict.get(i));
        /*
        Everything seems to be in working order, on a very basic
        test. However you should check out the Dictionary Java
        has to offer.
        */
    }
}

Output:

Mapping Dictionary
[3 : Three]
[2 : Two]
[1 : One]
Testing get
dict[1]: One
dict[2]: Two
dict[3]: Three
TreeNode.java

class TreeNode {

    TreeNode left; // left node
    int data;     // data item
    TreeNode right; // right node

    // Constructor: initialize data to d and make this a leaf node
    public TreeNode( int d )
    {
        data = d;
        left = right = null; // this node has no children
    }

    // Insert a TreeNode into a Tree that contains nodes.
    // Ignore duplicate values.
    public void insert( int d )
    {
        if ( d < data ) {
            if ( left == null )
                left = new TreeNode( d );
            else
                left.insert( d );
        }
        else if ( d > data ) {
            if ( right == null )
                right = new TreeNode( d );
            else
                right.insert( d );
        }
    }
}
public class BinaryTree {
    private TreeNode root;

    // Construct an empty Tree of integers
    public Tree() { root = null; }

    // Insert a new node in the binary search tree.
    // If the root node is null, create the root node here.
    // Otherwise, call the insert method of class TreeNode.
    public void insertNode( int d ) {
        if ( root == null )
            root = new TreeNode( d );
        else
            root.insert( d );
    }

    // Preorder Traversal
    public void preorderTraversal() {
        preorderHelper( root );
    }

    // Recursive method to perform preorder traversal
    private void preorderHelper( TreeNode node ) {
        if ( node == null )
            return;

        System.out.print( node.data + " " );
        preorderHelper( node.left );
        preorderHelper( node.right );
    }
}
Inorder Traversal
public void inorderTraversal()
{	inorderHelper( root );
}

// Recursive method to perform inorder traversal
private void inorderHelper( TreeNode node )
{
    if ( node == null )
        return;

    inorderHelper( node.left );
    System.out.print( node.data + " " );
    inorderHelper( node.right );
}

Postorder Traversal
public void postorderTraversal()
{
    postorderHelper( root );
}

// Recursive method to perform postorder traversal
private void postorderHelper( TreeNode node )
{
    if ( node == null )
        return;

    postorderHelper( node.left );
    postorderHelper( node.right );
    System.out.print( node.data + " " );
}
public class BinaryTreeDemo{
    public static void main( String args[] )
    {
        Tree tree = new Tree();
        int intVal;

        System.out.println( "Inserting the following values: " );

        for ( int i = 1; i <= 10; i++ ) {
            intVal = ( int ) ( Math.random() * 100 );
            System.out.print( intVal + " " );
            tree.insertNode( intVal );
        }
    }
}
System.out.println("Preorder traversal");
tree.preorderTraversal();

System.out.println("Inorder traversal");
tree.inorderTraversal();

System.out.println("Postorder traversal");
tree.postorderTraversal();

Conclusion

In conclusion, we have only scratched the surface with these examples. However, now you have a great base on which to build. So get out there and start building, breaking, and having fun with your own code. Please check out my website for more information:

tfoss0001.github.io
For fun and interactive video tutorials check out my youtube channel:

DoEasy Productions

There are playlists on that channel to help guide you through these examples and to give you ideas for your own programs. Thank you very much.

-Torin Foss