Intended audience: Student who has working knowledge of Python

Target compiler: I’ll try to center the discussion on C99 using gcc 7.4

- Code examples might have an accompanying link
  - Follow link to step through example at pythontutor.com
  - Does a nice job of graphically showing variables in memory, the heap, and the stack

- In the interest of clarity, avoiding clutter, many examples lack a NOMEM check
  - *Never* assume `malloc`, etc., succeeded
Linked Lists

• Pointers to heap memory are well-suited to working with linked lists

• A List:
  • Linear, homogeneous container
  • *Not* indexed (no constant-time access)
  • Middle can be modified in constant time

• Made of *nodes*
  • Container for single element
  • Has references (pointers) to other nodes
Lists

- A sequence of elements
- *Not* indexable (immediately)
  - To access 5\textsuperscript{th} element, must visit the preceding 4
- Space is allocated for each new element
- Consecutive elements are linked together with a pointer
- Middle can be modified in constant time
Lists as Ordered Pairs

- For languages w/out explicit pointers, such as Bash, Maple, Python, and Java, it might be helpful to consider a list as an ordered pair
  1. The item (payload)
  2. The rest of the list

\[(\alpha, (\beta, (\gamma, (\delta, ()))))\]

- Where () is the empty list
- We might use a class
- Or, simply, nested arrays, of size 2 (or, empty)
  - This is a very LISP notion
Time, Operations on Lists

- Access (same as searching) – linear time ($\Theta(n)$)
- Modifying anywhere – constant time ($\Theta(1)$)
- Inserting
  - At front – $\Theta(1)$
  - Append – $\Theta(n)$, unless pointer to last element kept
In Python
• Python’s list is really an array
  • Really? How might we tell?
  • In any case, elements are accessed in constant time
• We’ll use Python’s list to hold our duples
  • We’ll call them cells, or nodes
• Let the empty list, [], be an empty list
• Remember, everything in Python is a reference (pointer)
```
L = []
    # add 24 to front
L = [ 24, L ]
print L
    # add 3 to the front
L = [ 3, L ]
print L
```

Would output:
```
[ 24, [] ]
[ 3, [ 24, [] ]]
```
def append( L, e ) :
    '''Append item e to end of L
    Note, reference L doesn’t change’’’

    t = L    # start at beginning

    while t != [] :
        t = t[1] # move to next cell

    # We have our hands on the last cell (empty list)

    # Make it a pair, w/a new end-of-list
    t.extend( [ e, [] ] )
Searching a List in Python

```python
def search( L, t ):
    '''Return cell of L that contains t, None if not found'''

    while L != [] :
        if L[0] == t :
            return L
        L = L[1]  # move to next cell

    return None  # didn't find it
```

Kurt Schmidt (Skipjack Solutions)  
Linked Lists in C  
November 1, 2021
Map – Apply Function to a List

```python
def apply( L, fn ):
    while L != [] :
        fn( L )
        L = L[1]  # move to next cell
```

`fn` is any function that takes a single cell, modifies it. E.g.:

```python
def square( c ):
    c[0] *= c[0]
```
Examples of Apply

Given:

\[ L = [ 1, [ 2, [ 3, [] ]]] \]

Print the list:

```python
def printCell( cell ):
    print cell[0]
apply( L, printCell )
```

```
1 2 3
```

```python
apply( L, square )
apply( L, printCell )
```

```
1 4 9
```
In C
 typedef struct sNode sNode;
 struct sNode {
   sNode *next;
   other data fields...
 };

• Such a node makes up a *singly-linked* list
  • Can only be traversed in one direction
  • We could make a *doubly-linked* list

• The `typedef` allows us to refer to the type simply as `sNode`
```c
#include <stdio.h>
#include <stdlib.h>

typedef struct sNode sNode;

struct sNode
{
    sNode *next;
    int data;
};

void print( sNode *l, FILE *fp )
{
    fprintf( fp, "<" );
    while( l != NULL )
    {
        fprintf( fp, " %d", l->data );
        l = l->next;
    }
    fprintf( fp, " >" );
}
```
```c
int main( int argc, char *argv[] )
{
    sNode *l = NULL ,
        *t = NULL ;

    l = (sNode*) malloc( sizeof( sNode ) ) ;
    l->data = 13 ; l->next = NULL ;

    t = (sNode*) malloc( sizeof( sNode ) ) ;
    t->data = 12 ; t->next = l ; l = t ;

    t = (sNode*) malloc( sizeof( sNode ) ) ;
    t->data = 5 ; t->next = l ; l = t ;

    print( l, stdout ) ;
    printf( "\n" ) ;

    return 0 ; /* memory leak */
}
```

https://goo.gl/SHQHrK
Freeing a Linked List

Never do this:

```c
sNode *p ;
for( p=L; p!=NULL; p=p->next )
    free( p ) ;
```

- This is the source of a subtle, but sinister, bug
- We free memory, and then we look at it

Consider this:

```c
sNode *p=L; *q ;
while( p != NULL )
{
    q = p->next ;
    free( p ) ;
    p = q ;
}
```
typedef struct sNode sNode;

struct sNode { /* a node (cell) in a singly-link list */
    int data; /* the payload */
    sNode* next;
};

/* Wrap an item in a node (cell) */
sNode* newNode( int d ) {
    sNode *newp;
    newp = (sNode*) malloc( sizeof( sNode ) );
    if( newp != NULL ) {
        newp->data = d;
        newp->next = NULL;
    }
    return newp;
}

typedef sNode* List;
/* addfront: add newp to front of listp *
 * return ptr to new list        */
sNode* addfront( sNode *listp, sNode *newp )
{
    newp->next=listp ;
    return newp ;
}
list = addfront( list, newNode( 13 )) ;
list = addfront( list, newNode( 12 )) ;
list = addfront( list, newNode( 5 )) ;

List would be ( 5 12 13 )
/* append: add newp to end of listp *
* return ptr to new list       */
sNode* append( sNode* listp, sNode* newp )
{
    sNode *p ;
    if( listp == NULL )
        return newp ;
    for( p=listp; p->next!=NULL; p=p->next )
        /* Find last node */
    p->next = newp ;
    return listp ;
}

list = append( list, newNode( 42 )) ;

List would be ( 5 12 13 42 )
/* lookup: linear search for t in listp    *
 * return ptr to node containing t, or NULL */
sNode* lookup( sNode *listp, int t )
{
    for( ; listp != NULL; listp = listp->next )
        if( listp->data == t )
            return listp ;
    return NULL; /* no match */
}