Lec_24.md

# Lecture 24 - Linked Lists and Recursion

Nov. 04/2020

## Linked Lists

From previously, we built the **Linked List** class definitions:

#### listNode

```
class listNode {
  private:
    int key;
    listNode* next;
  public:
    listNode();
    listNode(int k);
    listNode(int k, listNode* n);
    listNode(const listNode& other);
    ~listNode();
    int getKey() const;
    listNode* getNext() const;
    void setKey(int k);
    void setNext(listNode* n);
    void print() const;
};
```

#### linkedList

```
class linkedList {
  private:
    listNode* head;
  public:
    linkedList();
    linkedList(const linkedList& other);
    ~linkedList();
    linkedList & operator=(const linkedList& rhs);
    bool insertKey(int k);
    bool deleteKey(int k);
    bool keyExists(int k);
    void print() const;
};
```

### linkedList Copy Constructor

```
linkedList::linkedList(){
  head = NULL;
}

linkedList(const linkedList& other){
  listNode* ptr = other.head;
  listNode* nptr = NULL;
  head = NULL;
  while(ptr != NULL) {
```

```
    nptr = new listNode(ptr->getKey());
    // insert *nptr at end of list
    ptr = ptr->getNext();
  }
}
```

Notes:

1. `linkedList::linkedList()`

- Create a new **linked list**, with head `NULL`

2. We want the **copy constructor** to have a **deep** implementation

- If we copy a list, we want to keep all the *next* nodes
- But don't want to have error of shared object data

3. `linkedList(const linkedList& other)`

Alternatively,

```
linkedList::linkedList(const linkedList& other) {
  listNode* ptr = other.head;
  listNode* last = NULL;
  listNode* nptr = NULL;
  head = NULL;
  while(ptr != NULL) {
    nptr = new listNode(ptr->getKey());
    if (last == NULL) head = nptr;
    else last->setNext(nptr);
    ptr = ptr->getNext();
    last = nptr;
  }
}
```

## linkedList Destructor

If we don't write a destructor, the default one is given

- `listNode* head` is deleted
  - Results in **memory leaks**

So we must implement the destructor **deeply**

```
linkedList::~linkedList() {
  listNode* ptr;
  while(head != NULL) {
    ptr = head;
    head = ptr->getNext();
    delete ptr;
  }
}
```

## linkedList operator=

If we don't write an overload for the `operator=` operator

- C++ provides a **shallow** implementation
  - Results in *shared object data*

So we must also implement the `operator=` **deeply**

```
linkedList& linkedList::operator=(const linkedList& rhs) {
  if (this == &rhs) return (*this);
  // Delete the list of *this, see destructor’s code
  // Allocate new listNode’s and copy from rhs, see copy
  // constructor’s code
  return (*this);
}
```

Notes:

1. `if (this == &rhs) return (*this)`

- Prevents deletion of current object

## Recursion

**Recursion** is a programming mechanism for implementing **divide-and-conquer** algorithms

- Implemented using **recursive** methods/functions
  - **recursive** methods/functions are those that call themselves
- Will examine:
  - **Recursive definition**
  - **Recursive functions**
  - Tracing **recursive functions**
  - How to use **recursion** to solve a problem

### Recursive Definition

How do you calculate a factorial?

```
f(n) = 1        , if n = 1 //basis
f(n) = n*f(n-1) , if n > 1 //recursive
```

Notes:

1. The **basis**, or the **base case** is the simplest version of the problem
2. The **recursive** portion is the **recursive** call that runs

- This divides the problem until reaching the **basis**

Implementation:

```
int factorial (int n) {
  if (n == 1) return (1);
  return (n*factorial(n-1));
}
```

Notes:

1. Notice that this implementation models the mathematical function above
2. `return n*(factorial(n-1))`

- Returning a call to the function itself (**recursive call**)

### Recursive Process

Lets trace out the recursive calls (**Recursive Trace**)

#### Downwards

```
//for example, calculate 4! (or factorial(4))
int factorial (int n) { // n = 4
  if (n == 1) return (1);
  return (n*factorial(n-1));
}
int factorial (int n) { // n = 3
  if (n == 1) return (1);
  return (n*factorial(n-1));
}
int factorial (int n) { // n = 2
  if (n == 1) return (1);
  return (n*factorial(n-1));
}
int factorial (int n) { // n = 1
  if (n == 1) return (1); // returns n = 1 here
  return (n*factorial(n-1));
}
```

Notes:

1. There are 4 **stack frames** created here

- A **stack frame** can be thought of a "copy" of a function in memory
  - The **stack frames** are written out above, but ***don't*** exist in code

#### Upwards

```
//using same example above
//the last function that ran was factorial(1)
//continuation of above
int factorial (int n) {// n = 1
  if (n == 1) return (1);// returns n = 1 here
  return (n*factorial(n-1));
}
int factorial (int n) {// n = 2
  if (n == 1) return (1);
  return (n*factorial(n-1));// returns (n=2)*(1) = 2
}
int factorial (int n) { // n = 3
  if (n == 1) return (1);
  return (n*factorial(n-1));// returns (n=3)*(2) = 6
}
int factorial (int n) { // n = 4
  if (n == 1) return (1);
  return (n*factorial(n-1));// returns (n=4)*(6) = 24
}
```

So finally, `factorial(4)` returns 24

- Which is the value of 4!

## Thinking Recursively

Given a problem, what steps are there to develop a recursive solution?

1. Find the **Basis**

- Identify the simplest size of the problem you can solve

2. Define the **Recursion**

- For any case larger than the **basis**
  - Think about dividing the problem into parts that look like the original problem, but smaller in size
  - Goal is to reach the basis

3. Write the Code
4. **Trace** the Code to validate

## Summing an Array

1. Define **basis**:

- The sum of an array with size 1 is
  - Just the value of that element

2. Define **recursive**

- The sum of an array with size 2 is
  - The value of the left element + the *sum of the array on the right*
    - `sum(a[0:n]) = sum(a[0]) + sum(a[1:n])`
    - In this case, `sum(a[0]) = a[0]`
      - Same process as **basis**
  - The *sum of the array on the right* is the **basis**
- Expanding to higher dimensions,
- The sum of an array with size n is
  - The value of the left element + the *sum of the array on the right*
    - `sum(a[0:n]) = sum(a[0]) + sum(a[1:n])`
    - `sum(a[1:n]) = sum(a[1]) + sum(a[2:n])`
    - `sum(a[2:n]) = sum(a[2]) + sum(a[3:n])`
    - ...
    - `sum(a[n]) = sum(a[n]) = a[n]`
      - This is the **basis**

Implementation:

#### sum_array

```
int sum_array(int* a, int left, int right) {
  if (left == right) return a[left];
  else return a[left]+ sum_array(a, left+1, right);
}
```

## Tracing the Function

Given an array

> a = [5,3,7]

**Trace** `sum_array(a,0,2)` (sum a from index 0 to 2, or just sum all of a)

#### Downwards

```
//left = 0
//right = 2
int sum_array(int* a, int left, int right) {
  if (left == right) return a[left];
  else return a[left]+ sum_array(a, left+1, right); //recursive call
}

//left = 1
//right = 2
int sum_array(int* a, int left, int right) {
  if (left == right) return a[left];
  else return a[left]+ sum_array(a, left+1, right); //recursive call
}

//left = 2
//right = 2
int sum_array(int* a, int left, int right) {
  if (left == right) return a[left]; //now start traversing back up recursion
  else return a[left]+ sum_array(a, left+1, right);
}
```

#### Upwards

```
//left = 2
//right = 2
int sum_array(int* a, int left, int right) {
  if (left == right) return a[left]; //return 7
  else return a[left]+ sum_array(a, left+1, right); //recursive call
}

//left = 1
//right = 2
int sum_array(int* a, int left, int right) {
  if (left == right) return a[left];
  else return a[left]+ sum_array(a, left+1, right);
  //return a[1] + 7
  //a[1] + 7 = 3 + 7
  //return 10;
}

//left = 0
//right = 2
int sum_array(int* a, int left, int right) {
  if (left == right) return a[left]; //now start traversing back up recursion
  else return a[left]+ sum_array(a, left+1, right);
  //return a[0] + 10;
  //a[0] + 10 = 5 + 10;
  //return 15;
}
```

So finally, `sum_array(a,0,2)` returns 15!