CSCI 2270 – Data Structures

Recitation 3, Jan 2019

Dynamic Memory and Linked Lists

Objectives

1. Dynamic memory
2. Freeing memory
3. Destructors
4. Linked Lists Basics
5. Insertion
6. Traversal
7. Deletion
8. Exercise

1. Dynamic Memory

What is static memory allocation?
When we declare variables, we are preparing the variables that will be used. This way a compiler can know that the variable is an important part of the program. So, the compiler allocates spaces to all these variables during compilation of the program. **The variables allocated using static memory allocation will be stored in *STACK*.**

But wait, what if the input is not known during compilation? Say, you want to pass input in the command line arguments (During runtime). The size of the input is not known beforehand during compilation.

The need for dynamic memory!!
We suffer in terms of inefficient storage use and lack or excess of slots to enter data. This is when dynamic memory play an important role. This method allows us to create storage blocks or room for variables during run time. Now there is no wastage. **The variables allocated using dynamic memory allocation will be stored in *HEAP.***

Tip!
Fine dining restaurant vs drive thru
Most of the times we reserve space at a fine dining restaurant by calling them up or reserving online. (The space is wasted if you do not show up even after reserving). The restaurant can also not accomodate more than its capacity. (Static memory)

Consider McDonalds drive-thru, you don't reserve space, you will somehow find a spot in the queue and just manage to grab your order. Here you are not reserving any seats beforehand, but still you can manage to get a meal. The restaurant can serve multiple customers even if they have not reserved. (Dynamic memory)

Dynamic memory allocation example.

```
int main()
{
   // Dynamic memory allocation
   int *ptr1 = new int;
   int *ptr2 = new int[10];
}
```

2. Freeing Memory

Once a programmer allocates a memory dynamically as shown in the previous section, it is the responsibility of the programmer to delete/free the memory which is created. If not deleted/freed, even though the variable/array is still not used, the memory will continue to be occupied. This causes **memory leak.**

To delete a variable **ptr1** allocated dynamically

```
delete ptr1;
```

To delete an array ptr2 allocated dynamically

```
delete [] ptr2;
```

3. Destructors

Destructor is a member function which destructs or deletes an object.

Destructor is **automatically called** when any object defined in the program goes out of scope. An object goes out of scope when:

1. A function ends.
2. The program ends.
3. A block containing local variables ends.
4. A delete operator is called.

Destructors have same name as the class preceded by tilde(~). They don't take any argument and don't return anything.

Example usage of destructors.

```
class DestructorExample
{
    private:
        char *s;
        int size;

    public:
        DestructorExample(char *); // constructor
        ~DestructorExample();      // destructor
};

DestructorExample::DestructorExample(char *c)
{
    size = strlen(c);
    s = new char[size+1];
    strcpy(s,c);
}

DestructorExample::~DestructorExample()
{
    delete []s;
}
```

Questions:

1. What is the need of a custom destructor ?
2. In the exercise given in this recitation can you think of a situation (or in absence of which line) when there will be a memory leak ?

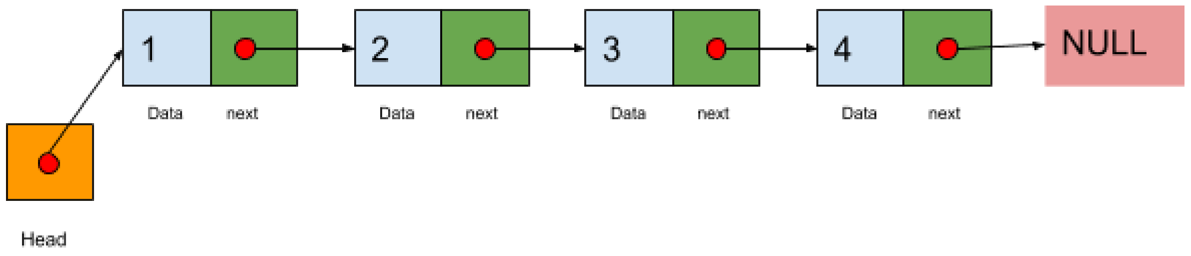
4. Linked List

A linked list is a data structure that stores a list, where each element in the list points to the next element.

Let's elaborate:

Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

- **Node** − Each Node of a linked list can store data and a link.
- **Link** − Each Node of a linked list contains a link to the next Node (unit of Linked List), often called 'next' in code.
- **Linked List** − A Linked List contains a connection link from the first link called Head. Every Link after the head points to another Node (unit of Linked List). The last node points to NULL.



In code, each node of the linked list will be represented by a class or struct. These will, at a minimum, contain two pieces of information - the data that node contains, and a pointer to the next node. Example code to create these is below:

```
class Node
{
public:
	int data;
	Node *next;
};
```

```
struct node
{
   int data;
   Node *next;
};
```

5. Insertion in a linked list

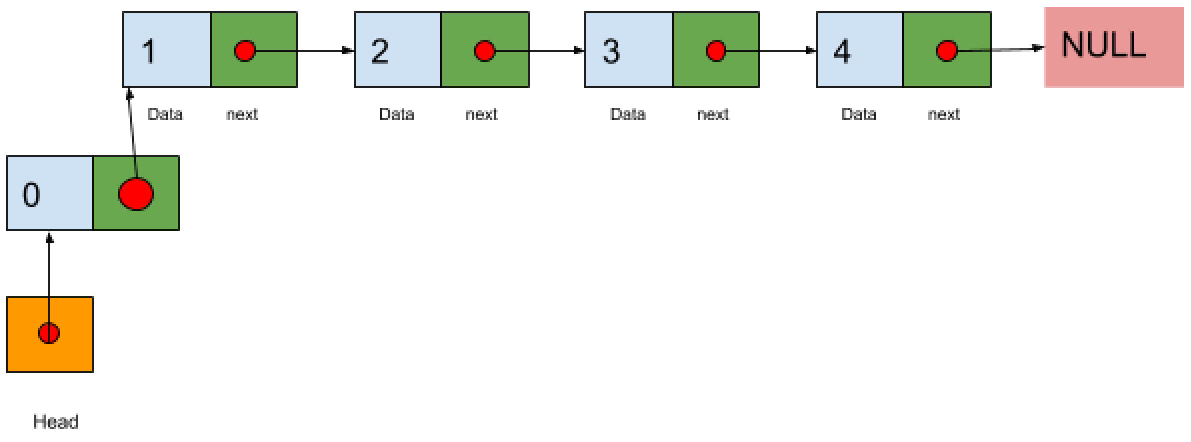
Adding a new node in a linked list is a multi-step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it must be inserted.

Scenarios in insertion

1. Insertion at the start.
2. Insertion at a given position.
3. Insertion at the end.

1. Inserting at the start of the list.

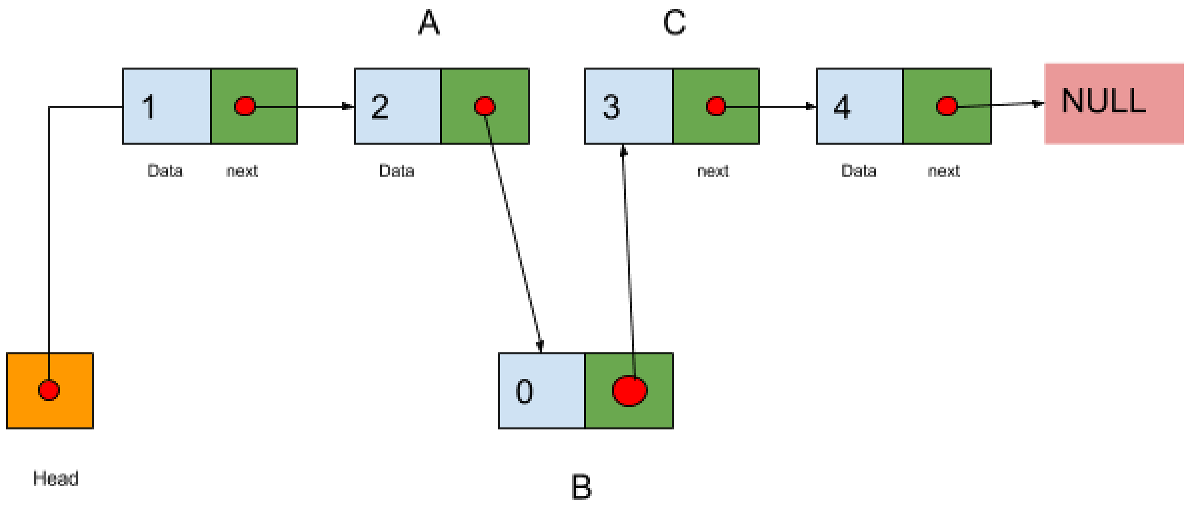
Now we will insert an element at the start of the list.



a. Create a new node,
b. Update the next pointer of that node to start of the list. (Value of the head pointer)
c. Update head pointer to new node.

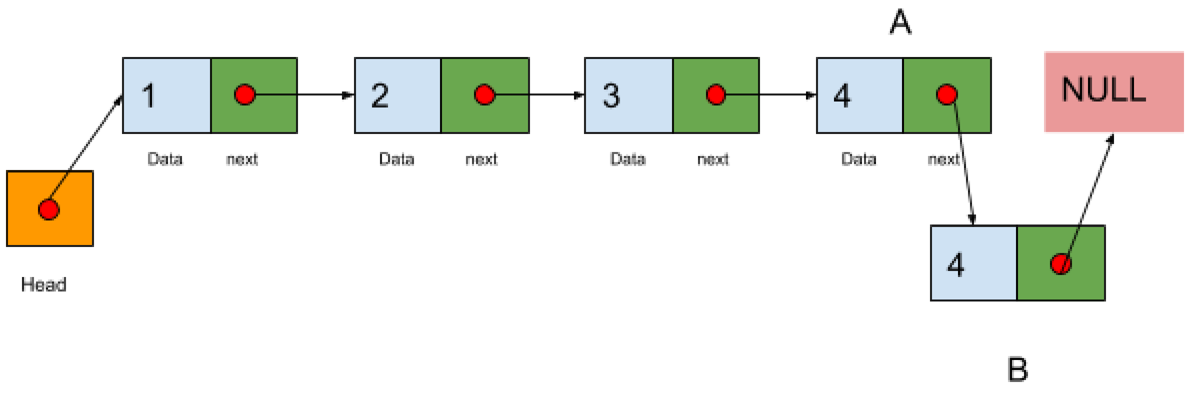
2. Insertion at a given position

For example let us insert a new node at position 2.



a. Create a new node (B).
b. Count and traverse until the node previous to given position i.e A.
c. Store the A’s next pointer value in a temporary variable.
d. Update the next pointer of A to the address of new node.
e. Copy the temporary variable’s value to B’s next pointer.
f. Now B’s next pointer points to the address of the node C.

3. Insertion at the end



a. Create a new node B.
b. Traverse till the node whose next pointer points to NULL. (A)
c. Update the next pointer of A to B’s address.
d. Point B’s next pointer to NULL.

6. Traversal and printing

To print a list, we need to traverse through all the nodes in the list until we encounter the last node. When a node points to NULL we know that it is the last node.

```
node = root;
while ( node != NULL )
 {
    cout << node->value << endl;
    node = node->next;
}
```

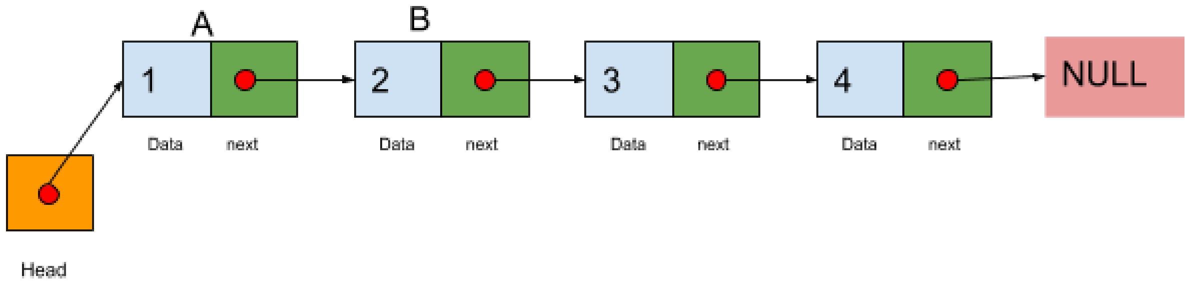
7. Deletion

Deleting a node in the linked list is a multi-step activity. Let's call the node to be deleted as 'A' and the node 'A' is pointing to as 'B'.

- First, the position of the node to be deleted ('A') must be found.
- The next step is to point the node pointing to 'A', to point to 'B'.
- The last step is to free the memory held by 'A'.

1. Deletion of the first node

a. Given below is the linked list representation before the deletion of the first node



b. Steps followed to delete the first node ('A') having value '1'.

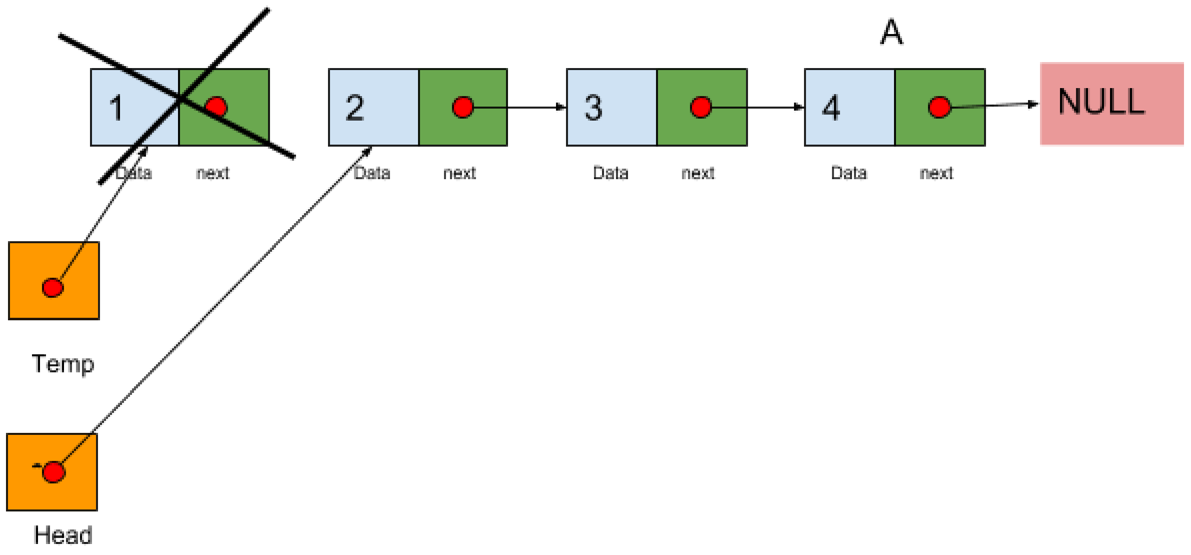
i. Create a variable **temp** having a reference copy of the head node.

ii. Point the head node from 'A' to 'B'

iii. Head is now pointing to 'B'. So, The Linked List's first element now is 'B' with the value'2'

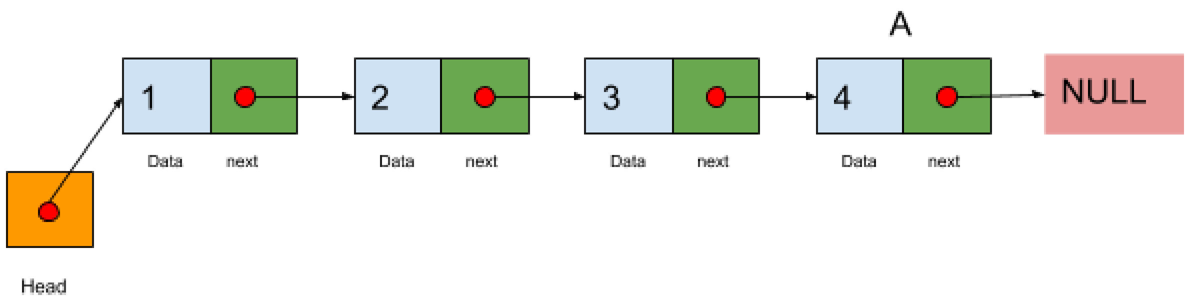
iv. Free the node 'A' pointed to by **temp**.

c. Linked list representation after deletion.



2. Deletion of the last node

a. Given below is a linked list representation before deletion of the last node

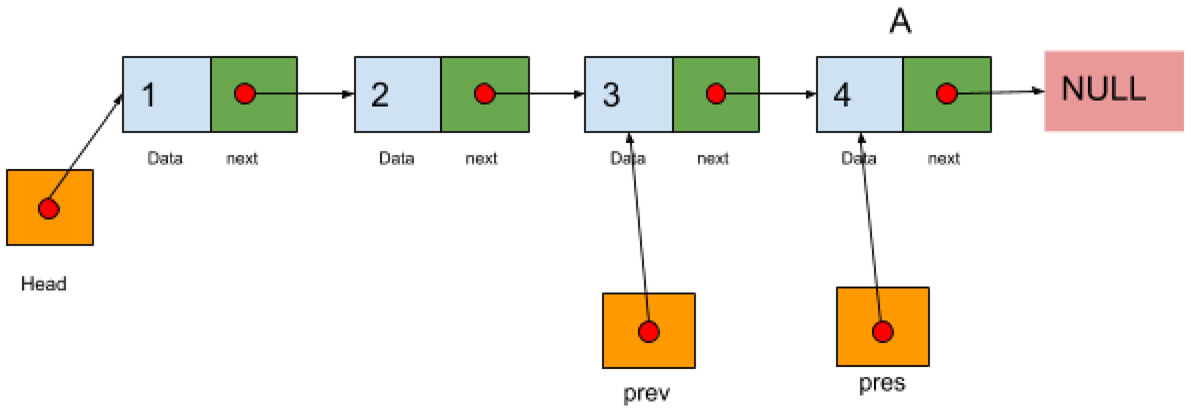


b. Steps followed to delete the last node ('A') having value '4'.

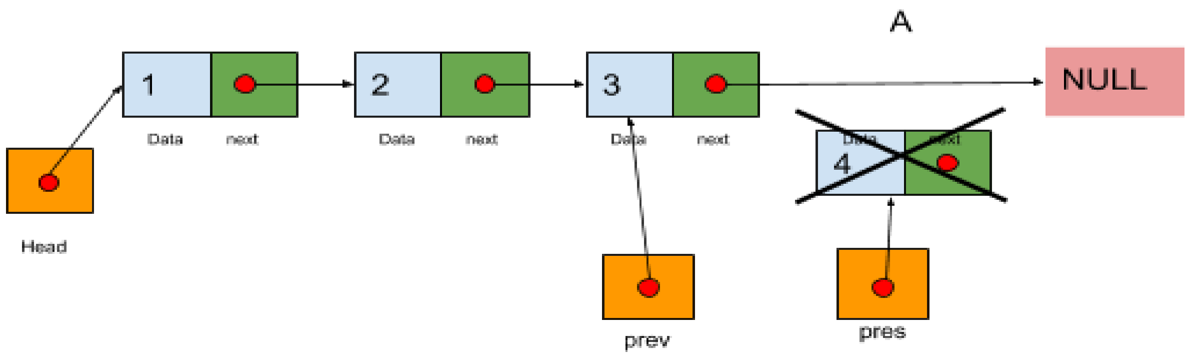
i. Create a variable **prev** having a reference copy of the head node.

ii. Create a variable **pres** having a reference copy of the next node after the head.
iii. Traverse the list until **pres** is pointing to the last node 'A'.
  prev will be pointing to the second last node now.
iv. Make **prev** point to **NULL.**
v. Free the node 'A' pointed to by **pres.**

c. Linked list representation after **prev** and **pres** have completed traversing.



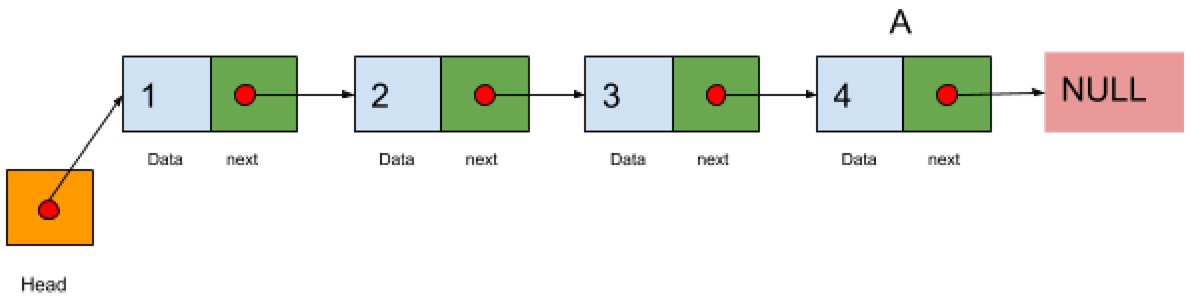
d. Linked list representation after deletion of the node 'A' and pointing prev to NULL.



3. Deletion of a linked list

The deletion of a linked list involves iteration over the complete linked list and deleting (freeing) every node in the linked list.

a. Given below is a linked list representation before deletion of the last node



b. Steps followed to delete every node in the linked list.

i. Create a variable **prev** having a reference copy of the head node.

ii. Create a variable **pres** having a reference copy of the next node after the head.

iii. While traversing the list, at each step delete/free the memory pointed to by **prev.**

iv. Now, point **prev** to the **pres** and point **pres** to the next node after **pres (ie. pres->next)**.

v. Traverse the list until **pres** is pointing to the **NULL**.
prev will be pointing to the second last node now.

vi. Free the memory pointed to by **prev.** Now every element in the linked list is deleted/freed.

c. Linked list representation after deletion of all the nodes.

8. Exercise

Open your exercise file and complete the TODOs in the C++ program which does the following:
1. It will read from a text file. Each line of the file contains a number. Provide the file name as a command line argument. Number of lines in the file is not known.
2. Create an array dynamically of a capacity (say 10) and store each number as you read from the file.
3. If you exhaust the array but yet to reach end of file dynamically resize/double the array and keep on adding.