Linked Lists

Chapter 17

Linked List

- A Linked List is a dynamically allocated data structure that is linked together in memory

Linked List

- A Linked List is a dynamically allocated data structure that is linked together in memory
- Linked lists have some advantages over an array or vector

Linked List

- A Linked List is a dynamically allocated data structure that is linked together in memory
- Linked lists have some advantages over an array or vector
  - They can grow and shrink as needed

Linked List

- A Linked List is a dynamically allocated data structure that is linked together in memory
- Linked lists have some advantages over an array or vector
  - They can grow and shrink as needed
  - The maximum size does not need to be known

Linked List

- A Linked List is a dynamically allocated data structure that is linked together in memory
- Linked lists have some advantages over an array or vector
  - They can grow and shrink as needed
  - The maximum size does not need to be known
  - Speed of insertion
    - Inserting into the middle of a populated vector is time consuming

Linked List Structure

- A Linked List consists of many nodes

Linked List Structure

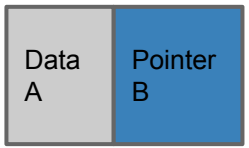
- A Linked List consists of many nodes

Linked List Structure

- A Linked List consists of many nodes
- Each node contains one or more members that hold data and a pointer to another node

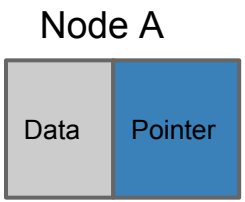
Linked List Structure

- A Linked List consists of many nodes
- Each node contains one or more members that hold data and a pointer to another node



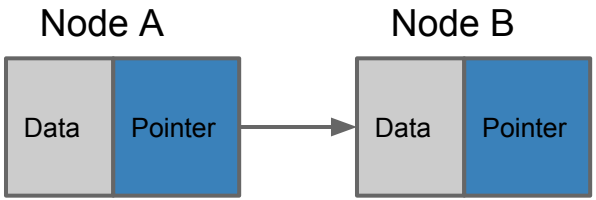
Linked List Structure

- Example



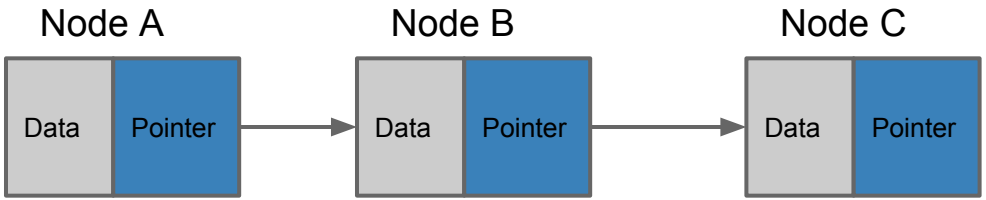
Linked List Structure

- Example



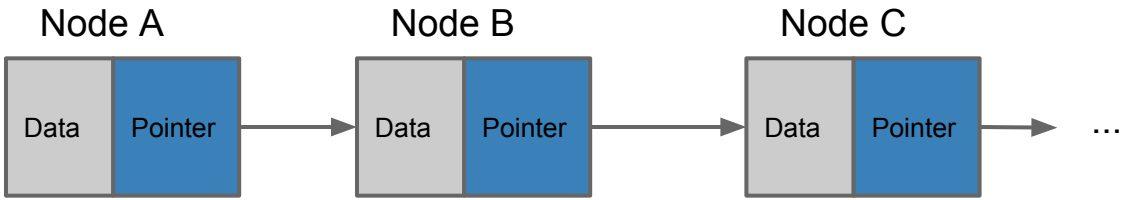
Linked List Structure

- Example



Linked List Structure

- Example

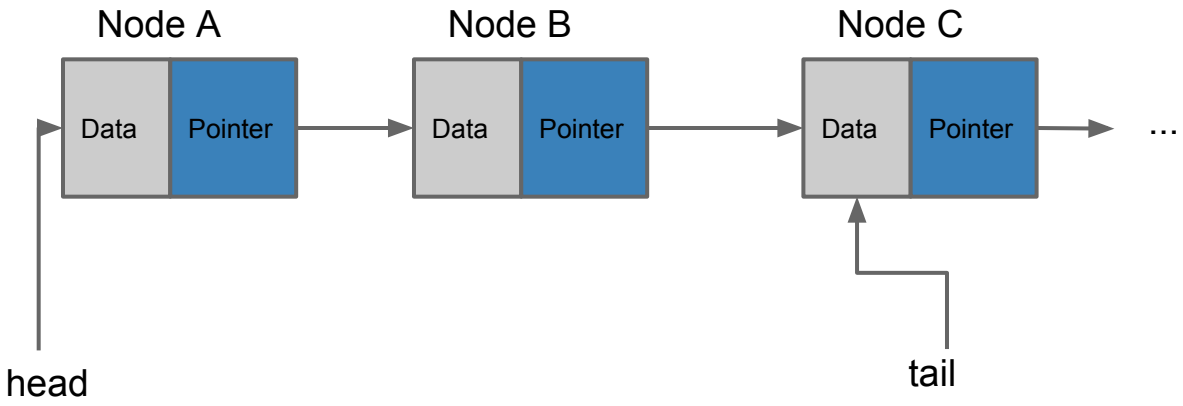


Linked List Structure

- A pointer to the first node in a non empty Linked List is called the ‘head’

Linked List Structure

- Example



Linked List Structure

- A pointer to the first node in a non empty Linked List is called the ‘head’
- All other nodes can be accessed through the head via iteration

Linked List Structure

- A pointer to the first node in a non empty Linked List is called the ‘head’
- All other nodes can be accessed through the head via iteration
- Each node points to the ‘next’ node in the list
  - The ‘head’ node points to node 2 which points to node 3 which points to ....

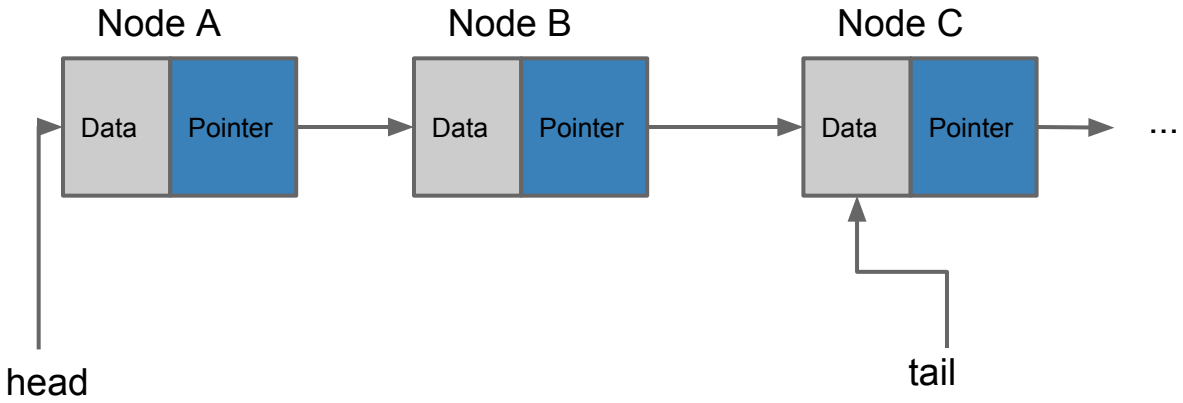
Linked List Structure

- A pointer to the first node in a non empty Linked List is called the ‘head’
- All other nodes can be accessed through the head via iteration
- Each node points to the ‘next’ node in the list
  - The ‘head’ node points to node 2 which points to node 3 which points to ....
  - The last node in the list will point to NULL

Linked List Structure

- Example

The linking of the nodes creates a chain of nodes that can be iterated



Linked List Structure

- A Linked List is created by placing all new nodes into the list in one of three locations

Linked List Structure

- A Linked List is created by placing all new nodes into the list in one of three locations
  - At the head of the list

Linked List Structure

- A Linked List is created by placing all new nodes into the list in one of three locations
  - At the head of the list
  - In between two existing nodes

Linked List Structure

- A Linked List is created by placing all new nodes into the list in one of three locations
  - At the head of the list
  - In between two existing nodes
  - At the end, or tail, of the list

Linked List Structure

- A Linked List is created by placing all new nodes into the list in one of three locations
  - At the head of the list
  - In between two existing nodes
  - At the end, or tail, of the list
- The choice of where to put the node depends on the structure
  - First In First Out (FIFO)
  - First In Last Out (FILO)

Linked List Structure

- To traverse a Linked List:

Linked List Structure

- To traverse a Linked List:
  - Create a node pointer and set it to the head node

Linked List Structure

- To traverse a Linked List:
  - Create a node pointer and set it to the head node
  - while the node you are looking at is not the node you wanted AND the node is NOT NULL
    - set node to node’s next pointer

Linked List Operations

- A Linked List has a set of standard operations

Linked List Operations

- A Linked List has a set of standard operations
  - Adding
  - removing
  - traversing
  - destroying

Linked List Operations - Adding

- There are three cases for adding a node to a linked list

Linked List Operations - Adding

- There are three cases for adding a node to a linked list
  - At the head
  - At the tail
  - Between two nodes

Linked List Operations - Adding

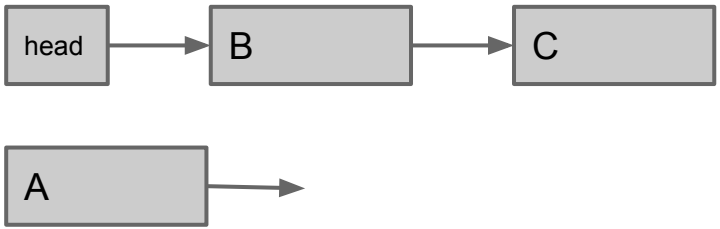
- There are three cases for adding a node to a linked list
  - At the head
  - At the tail
  - Between two nodes
    - Care must be taken not to lose access to nodes when performing this operation

Linked List Operations - Adding

- Adding at the head

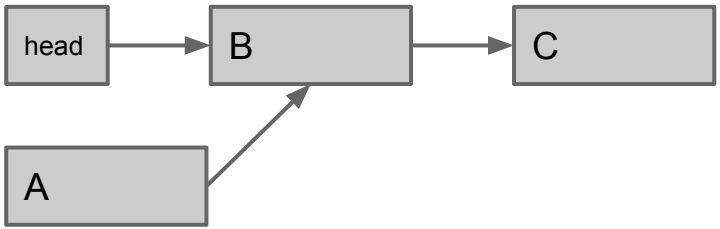
Linked List Operations - Adding

- Adding at the head



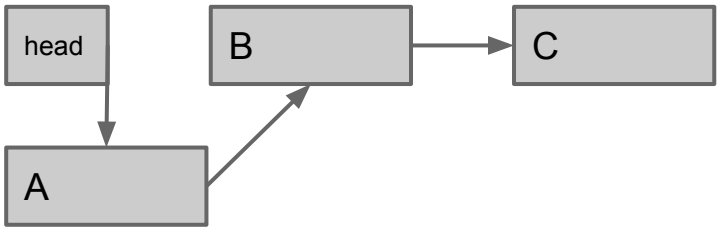
Linked List Operations - Adding

- Adding at the head
  - Set the new node’s next equal to the head node



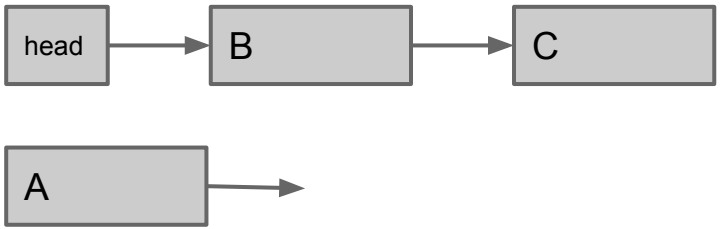
Linked List Operations - Adding

- Adding at the head
  - Set the new node’s next equal to the head node
  - Set the head pointer to the new node



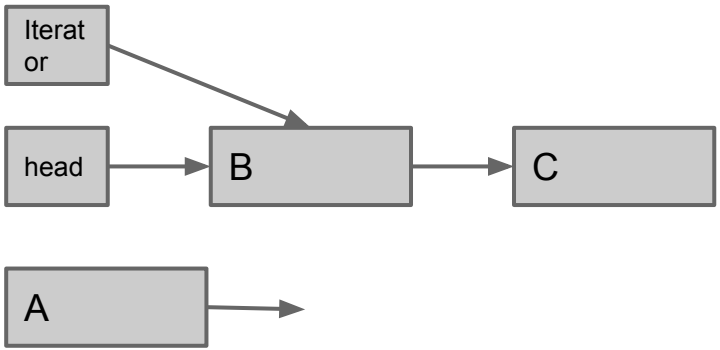
Linked List Operations - Adding

- Adding a node between two nodes



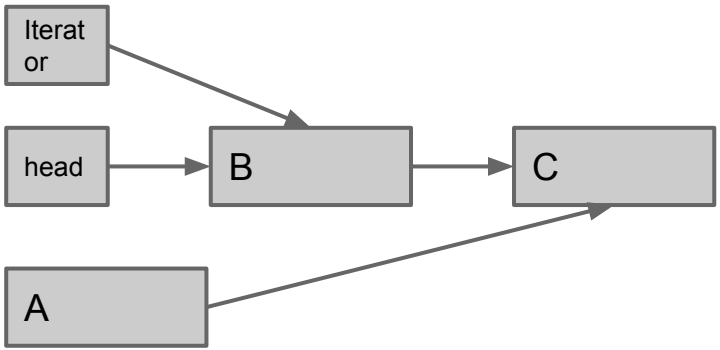
Linked List Operations - Adding

- Adding a node between two nodes
  - Iterate to the first node (node that will point to the new node)



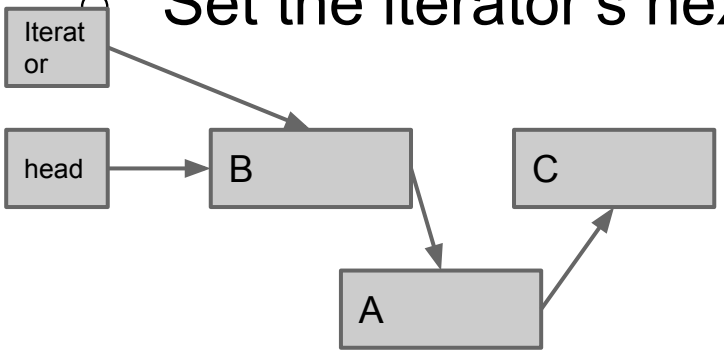
Linked List Operations - Adding

- Adding a node between two nodes
  - Iterate to the first node (node that will point to the new node)
  - Set the new node’s next to the iterator nodes next



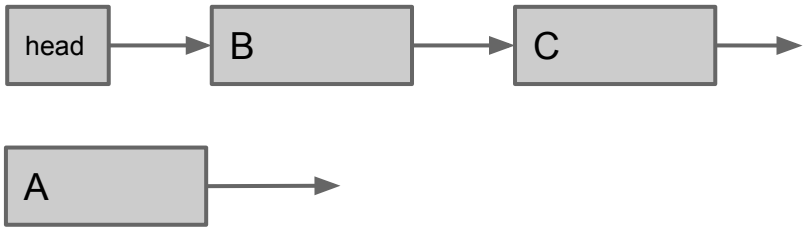
Linked List Operations - Adding

- Adding a node between two nodes
  - Iterate to the first node (node that will point to the new node)
  - Set the new node’s next to the iterator nodes next
  - Set the iterator’s next to the new node



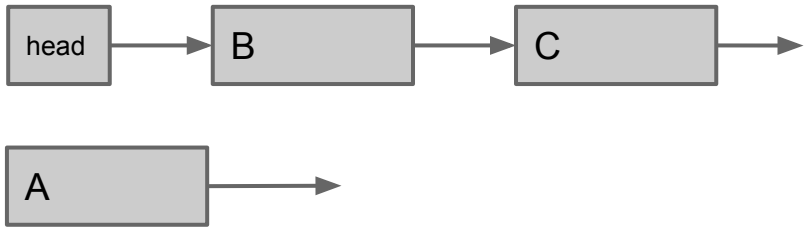
Linked List Operations - Adding

- How might we add a node at the head of the list?



Linked List Operations - Adding

- How might we add a node at the tail of the list?

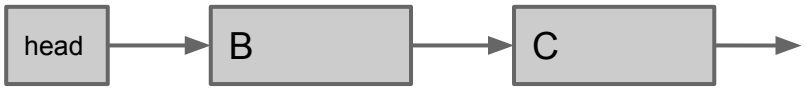


Linked List Operations - Deleting

- Deleting a node in a Linked List is very similar to adding

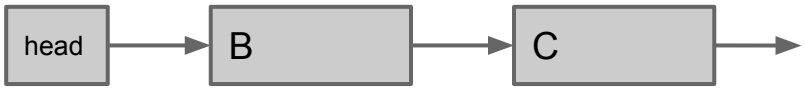
Linked List Operations - Deleting

- How might we delete a node at the head of the list?



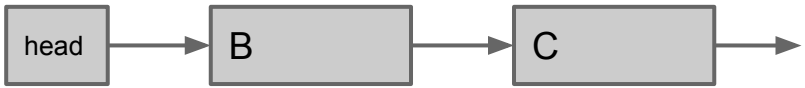
Linked List Operations - Deleting

- How might we delete a node at the head of the list?
  - What if it is the only node in the list?



Linked List Operations - Deleting

- How might we delete a node at the tail of the list?



Linked List Operations - Deleting

- How might we delete a node that lies between two nodes?

Linked List Operations - Destruction

- Destroying a Linked List can be tricky

Linked List Operations - Destruction

- Destroying a Linked List can be tricky
  - All nodes must be deleted without cutting off access to another node

Linked List Operations - Destruction

- Destroying a Linked List can be tricky
  - All nodes must be deleted without cutting off access to another node
  - If the nodes contain pointers, then that data must also be cleaned up

Linked List Operations - Destruction

- How might we clean up a Linked List such as this?

Linked List Template

- It is also possible to use templates in the Linked List implementation

Linked List Template

- It is also possible to use templates in the Linked List implementation
  - This is done just like a normal template class

Linked List Template

```
template<class T>
class LinkedListNode
{
        public:
                ...


        private
                T m_data;
                LinkedListNode* m_pNext;

};
```

Recursive Linked List Operations

- Recursion is a power tool, especially for Linked List functions

Recursive Linked List Operations

- Recursion is a power tool, especially for Linked List functions
- Most of the operations described before can be implemented using recursion

Recursive Linked List Operations

- How might we implement a ‘size’ function for determining the number of elements in the Linked List?

Recursive Linked List Operations

- How might we implement a ‘size’ function for determining the number of elements in the Linked List?

```
int size(LinkedListNode* pNode){
        if (pNode == NULL)
                return 0;
        else
                return 1 + size(pNode->getNext());
}
```

Recursive Linked List Operations

- How might we implement a ‘insert’ function for adding a node to the list?

Recursive Linked List Operations

- How might we implement a ‘remove’ function for adding a node to the list?

Recursive Linked List Operations

- How might we destruct our list using recursion?

Linked List Variations

- So far, the Linked List implementation has been a ‘singly linked list’

Linked List Variations

- So far, the Linked List implementation has been a ‘singly linked list’
  - This means that each node has a single pointer that points to the next node in the list

Linked List Variations

- So far, the Linked List implementation has been a ‘singly linked list’
  - This means that each node has a single pointer that points to the next node in the list
- There are other variations of a Linked List

Linked List Variations

- A ‘doubly Linked List’

Linked List Variations

- A ‘doubly Linked List’
  - Each node contains a pointer to the next node in the list

Linked List Variations

- A ‘doubly Linked List’
  - Each node contains a pointer to the next node in the list
  - Each node also contains a pointer to the previous node in the list

Linked List Variations

- A ‘circular Linked List’

Linked List Variations

- A ‘circular Linked List’
  - The same as a singly linked list, except the last node points to the first node

Linked List Variations

- A ‘circular Linked List’
  - The same as a singly linked list, except the last node points to the first node

- What problems might this cause on our function definitions, iterations, ...?