Chapter 4: Linked List

Department of Computer Science and Engineering
Kathmandu University

Contents

Linked List

Drawbacks of using sequential storage (to represent stacks and queues)

- A fixed amount of storage remains allocated (to the stack or queue) even when the structure is actually using a smaller amount or possible no storage at all.
- Size cannot be increased dynamically.

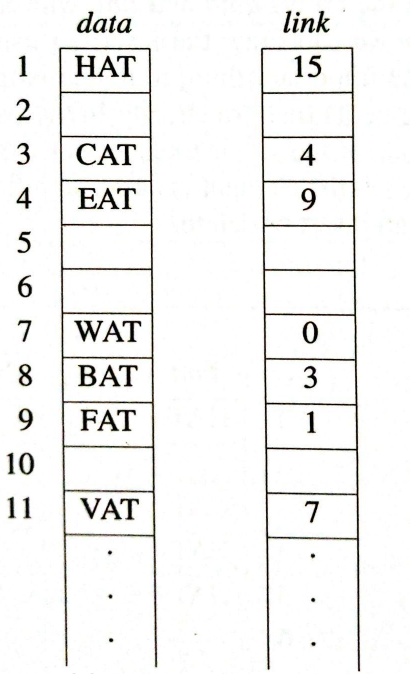
Solution:

- Using linked representation
  - Success elements in the list need not occupy adjacent space in memory.
  - To access list elements in the correct order, with each element the address of the next element in that list is stored.

Non-sequential list-representation

In a sequential representation, successive items of a list are located a fixed distance apart, i.e. the order of elements is the same as in the ordered list.

In a non-sequential/linked representation, items may be placed anywhere in memory.



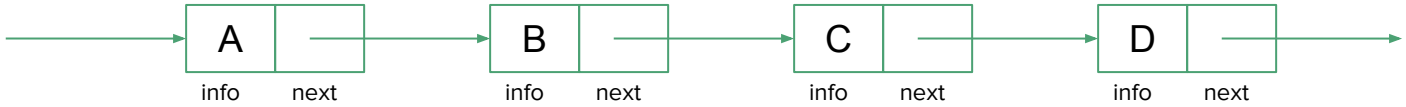
	data	*link*
1	HAT	15
2		
3	CAT	4
4	EAT	9
5		
6		
7	WAT	0
8	BAT	3
9	FAT	1
10		
11	VAT	7
	.	.
	.	.
	.	.

Non-sequential list-representation of the list of words: (BAT, CAT, EAT, FAT, HAT, ..., VAT, WAT)

Linked List

In general, a linked list is comprised of nodes; each node holding some information and pointers to another nodes in the list.

A node in a **singly linked list** has a link only to its successor in the sequence.



Linked List Operations

- Insertion
  - Insert a node at the beginning of the list (i.e., before the first node)
  - Insert a node at the end of the list (i.e., after the last node)
  - Insert a node after a particular node
- Deletion
  - Remove the first node
  - Remove the last node
  - Remove a node containing the given information
- Search: Check if the given information is present in the list
- Retrieve: Retrieve the node containing the given information
- Traversal: Visit all nodes in the list

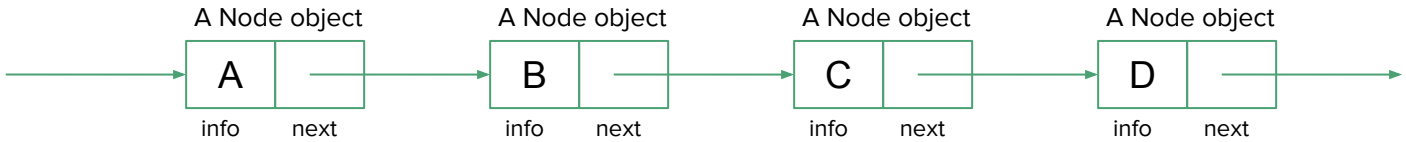
Linked List Implementation in C++

To implement a linked list in C++, we need

- A data structure to represent a node
  - For this we use self-referential structures/classes for defining a node's structure.
  - A self-referential structure is one in which one or more of its components is a pointer to itself.
- A pointer to identify the list (aka HEAD node)
  - Though a single pointer is needed, we can add metadata about the list.
  - We may also add a pointer to identify the end of the list.
- A mechanism to create new nodes
  - We use the `new` operator to create new nodes.
- A mechanism to remove nodes that are no longer needed
  - We use the `delete` operator.

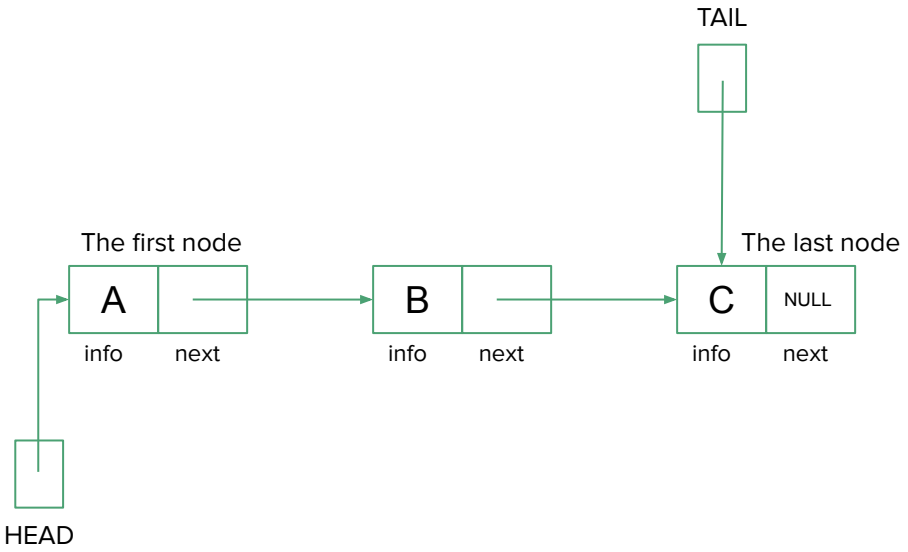
Linked List Implementation in C++

```
class Node {
public:
   int info;     // Data the node contains
   Node *next;   // Pointer to the next Node object in the chain
};
```



Linked List Implementation in C++

```
class List {
public:
   List();
   ~List();
   bool isEmpty();
   void addToHead(int data);
   void addToTail(int data);
   void add(int data, Node *predecessor);
   void removeFromHead();
   void removeFromTail();
   void remove(int data);
   bool search(int data);
   bool retrieve(int data, Node
*dataOutPtr);
   void traverse();
private:
   Node *HEAD; // Pointer to the first node
   Node *TAIL; // Pointer to the last node
};
```



Algorithms

Algorithm: List constructor
Input: A linked list, list(HEAD, TAIL)
Output: An empty list

Steps:

1. Initialize HEAD to NULL
2. Initialize TAIL to NULL

Algorithms

Algorithm: isEmpty
Input: A linked list, list(HEAD, TAIL)
Output: true if the list is empty, false otherwise

Steps:

1. If HEAD == NULL,
    a. return true
2. else
    a. return false
3. endif

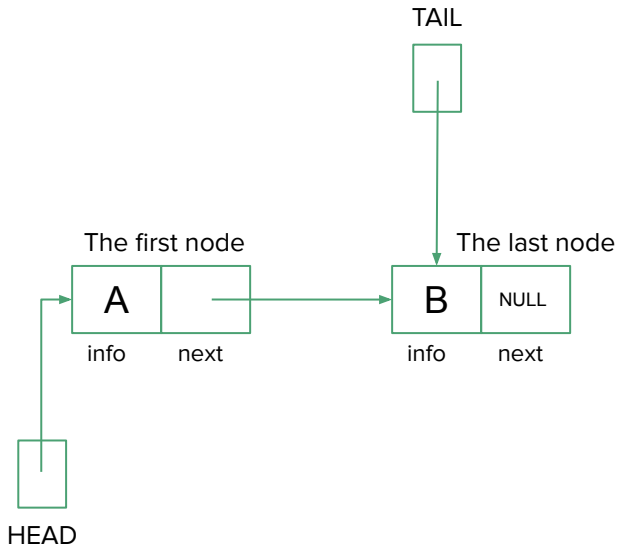
Algorithms

Algorithm: addToHead(data)

Input: A linked list, list(HEAD, TAIL), and the data to be added

Output: The updated list with data added to the list

Steps:



Algorithms

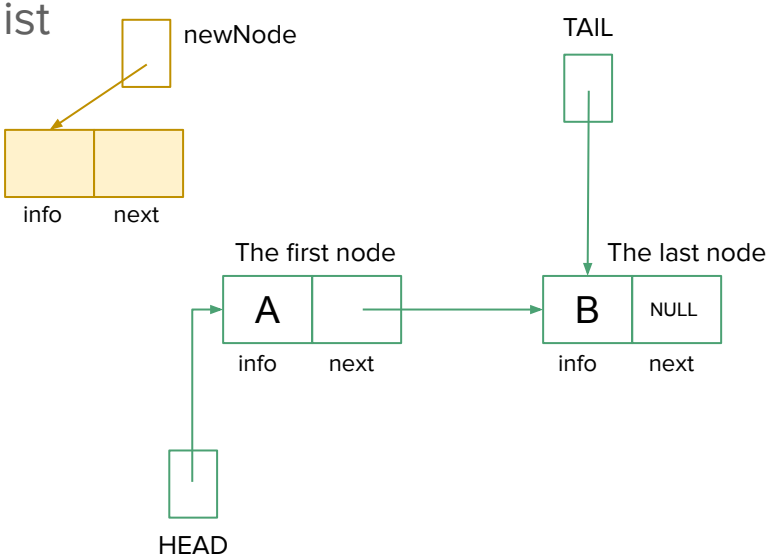
Algorithm: addToHead(data)

Input: A linked list, list(HEAD, TAIL), and the data to be added

Output: The updated list with data added to the list

Steps:

1. Create a new node, `newNode`



Algorithms

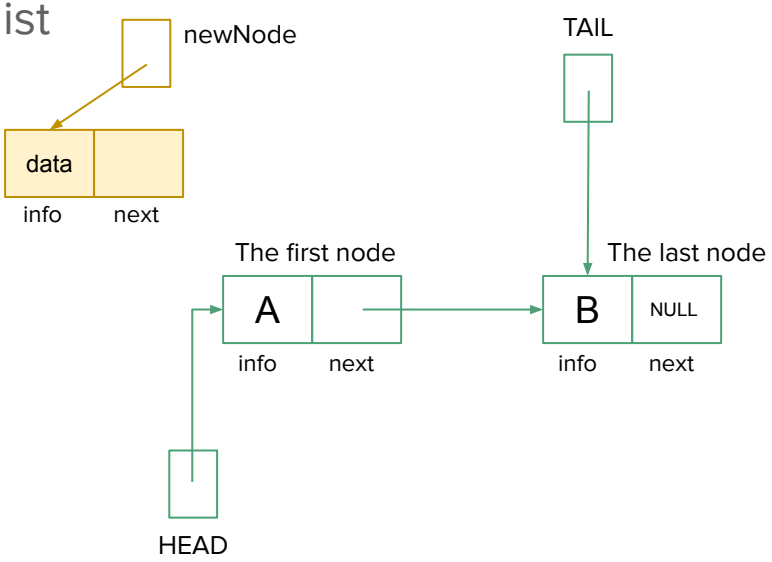
Algorithm: addToHead(data)

Input: A linked list, list(HEAD, TAIL), and the data to be added

Output: The updated list with data added to the list

Steps:

1. Create a new node, `newNode`
2. `newNode->info = data`



Algorithms

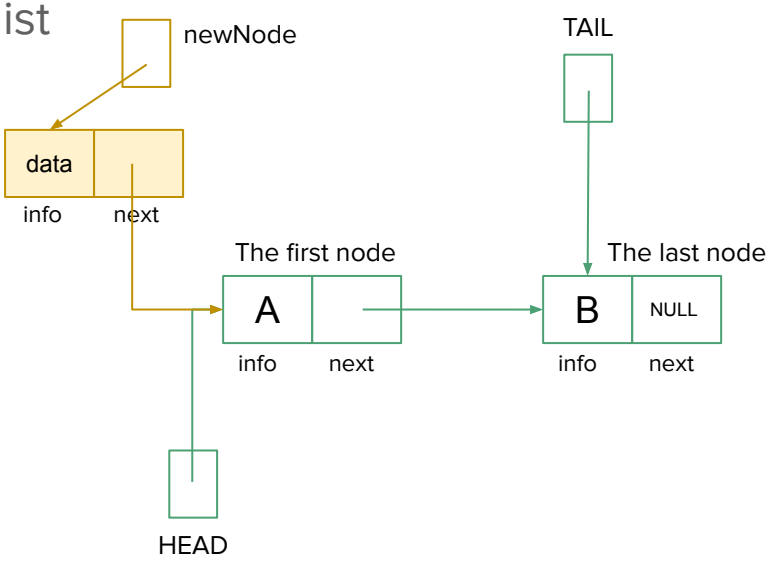
Algorithm: addToHead(data)

Input: A linked list, list(HEAD, TAIL), and the data to be added

Output: The updated list with data added to the list

Steps:

1. Create a new node, `newNode`
2. `newNode->info = data`
3. `newNode->next = HEAD`



Algorithms

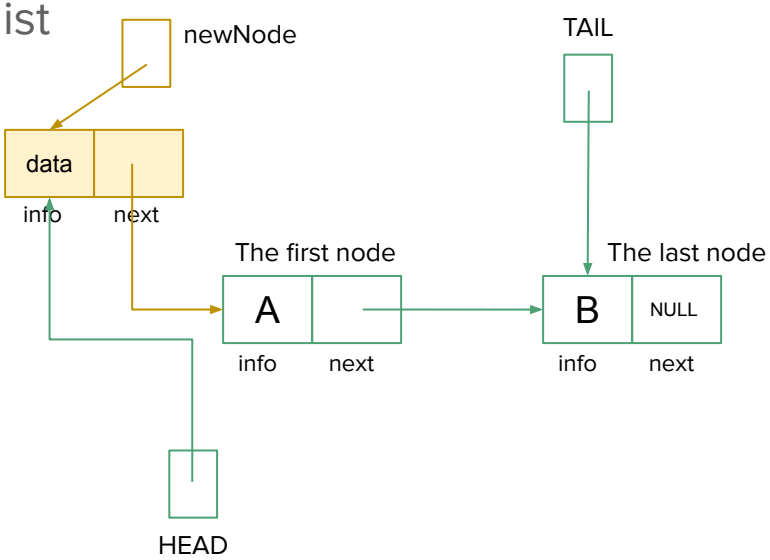
Algorithm: addToHead(data)

Input: A linked list, list(HEAD, TAIL), and the data to be added

Output: The updated list with data added to the list

Steps:

1. Create a new node, `newNode`
2. `newNode->info = data`
3. `newNode->next = HEAD`
4. `HEAD = newNode`

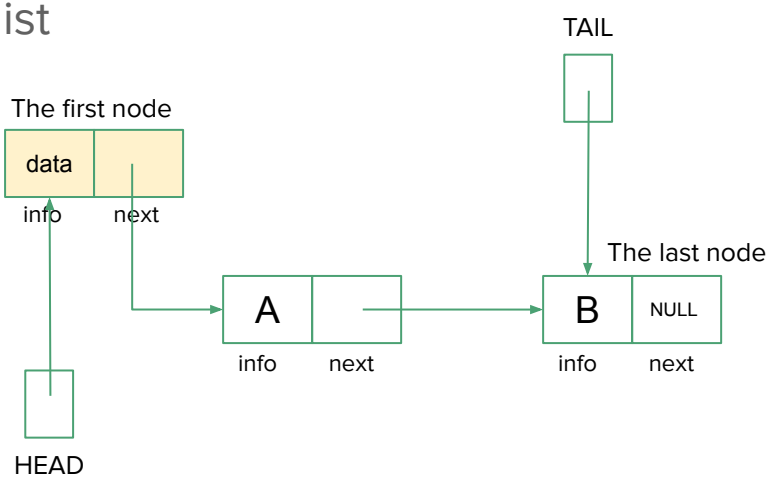


Algorithms

Algorithm: addToHead(data)
Input: A linked list, list(HEAD, TAIL), and the data to be added
Output: The updated list with data added to the list

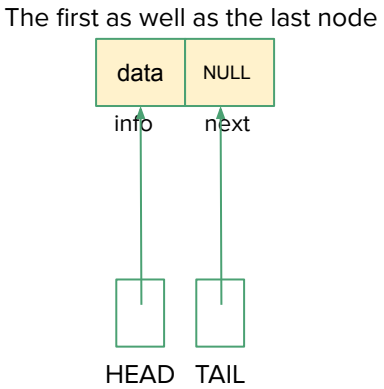
Steps:

1. Create a new node, `newNode`
2. `newNode->info = data`
3. `newNode->next = HEAD`
4. `HEAD = newNode`



Algorithms

Algorithm: addToHead(data)
Input: A linked list, list(HEAD, TAIL), and the data to be added
Output: The updated list with data added to the list



Steps:

1. Create a new node, `newNode`
2. `newNode->info = data`
3. `newNode->next = HEAD`
4. `HEAD = newNode`
5. If `TAIL == NULL` (i.e., when the added node is the only node in the list)
   a. `TAIL = HEAD`
6. endif

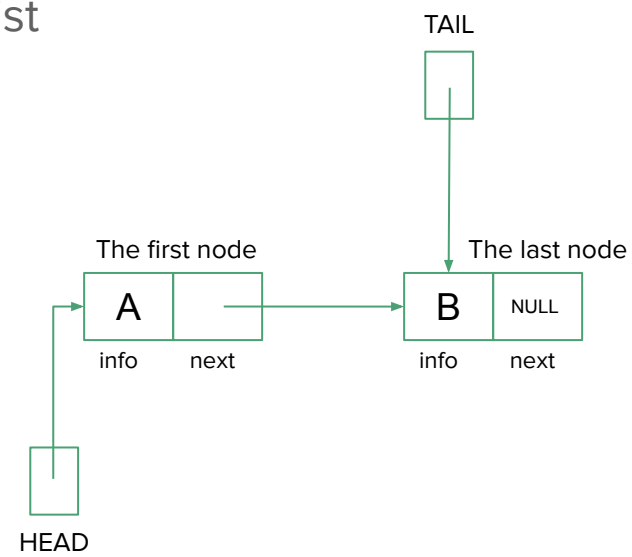
Algorithms

Algorithm: addToTail(data)

Input: A linked list, list(HEAD, TAIL), and the data to be added

Output: The updated list with data added to the list

Steps:

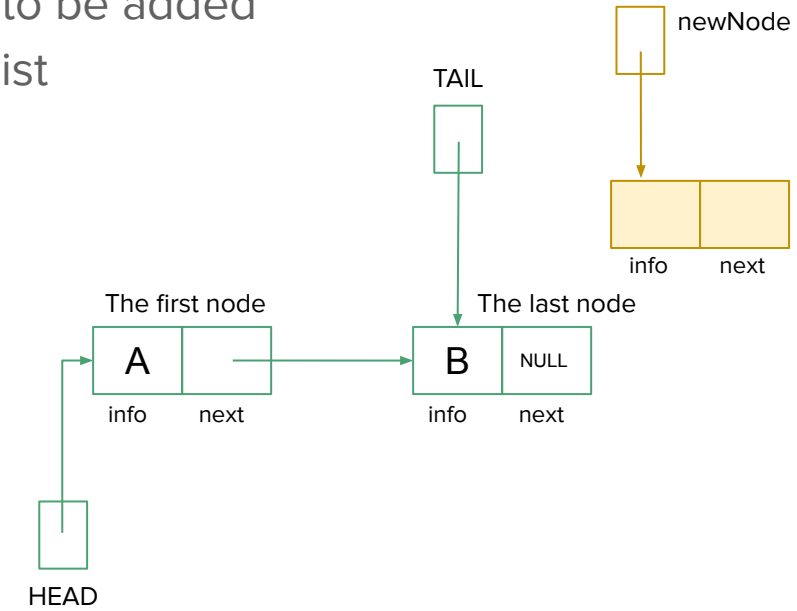


Algorithms

Algorithm: addToTail(data)
Input: A linked list, list(HEAD, TAIL), and the data to be added
Output: The updated list with data added to the list

Steps:

1. Create a new node, `newNode`



Algorithms

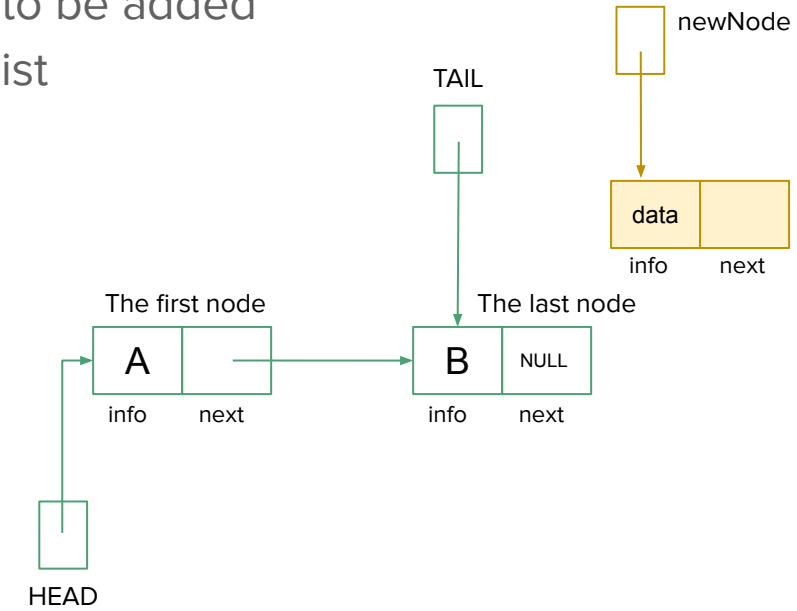
Algorithm: addToTail(data)

Input: A linked list, list(HEAD, TAIL), and the data to be added

Output: The updated list with data added to the list

Steps:

1. Create a new node, `newNode`
2. newNode->info = data

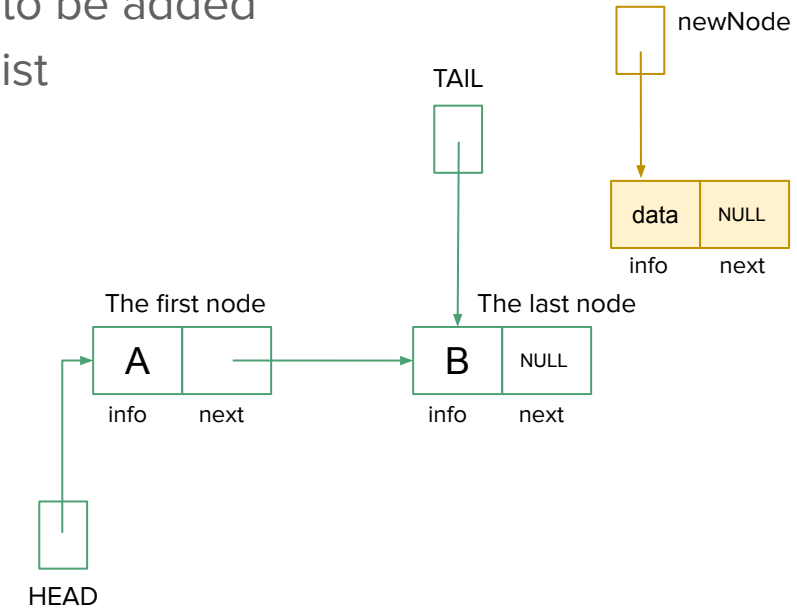


Algorithms

Algorithm: addToTail(data)
Input: A linked list, list(HEAD, TAIL), and the data to be added
Output: The updated list with data added to the list

Steps:

1. Create a new node, `newNode`
2. `newNode->info = data`
3. `newNode->next = NULL`

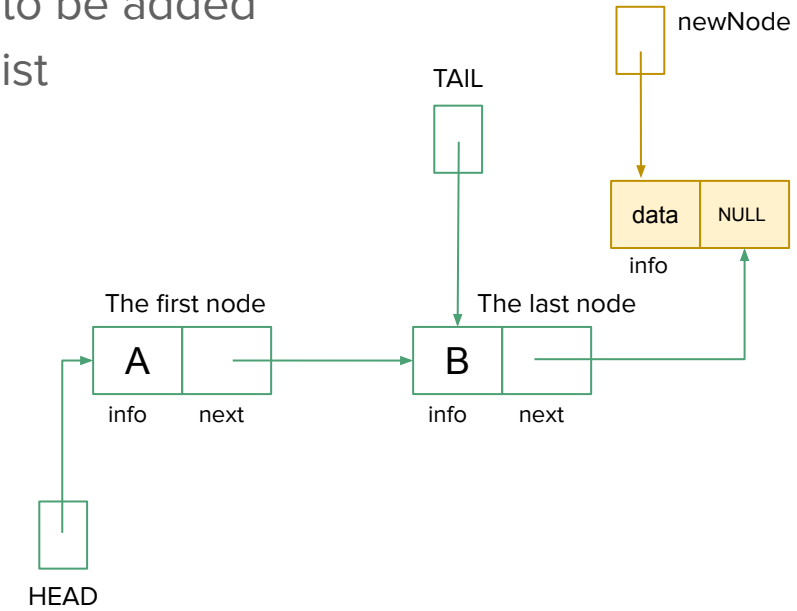


Algorithms

Algorithm: addToTail(data)
Input: A linked list, list(HEAD, TAIL), and the data to be added
Output: The updated list with data added to the list

Steps:

1. Create a new node, `newNode`
2. `newNode->info = data`
3. `newNode->next = NULL`
4. `TAIL->next = newNode`

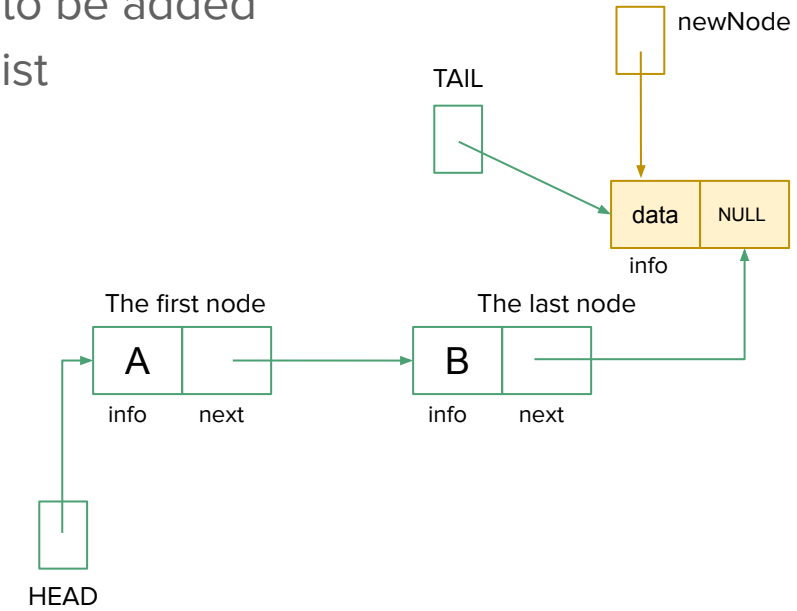


Algorithms

Algorithm: addToTail(data)
Input: A linked list, list(HEAD, TAIL), and the data to be added
Output: The updated list with data added to the list

Steps:

1. Create a new node, `newNode`
2. `newNode->info = data`
3. `newNode->next = NULL`
4. `TAIL->next = newNode`
5. `TAIL = TAIL->next`

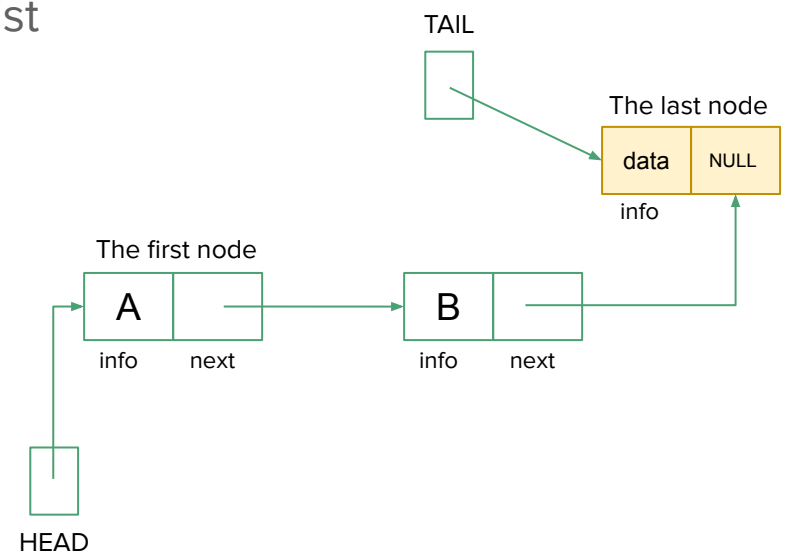


Algorithms

Algorithm: addToTail(data)
Input: A linked list, list(HEAD, TAIL), and the data to be added
Output: The updated list with data added to the list

Steps:

1. Create a new node, `newNode`
2. `newNode->info = data`
3. `newNode->next = NULL`
4. `TAIL->next = newNode`
5. `TAIL = TAIL->next`



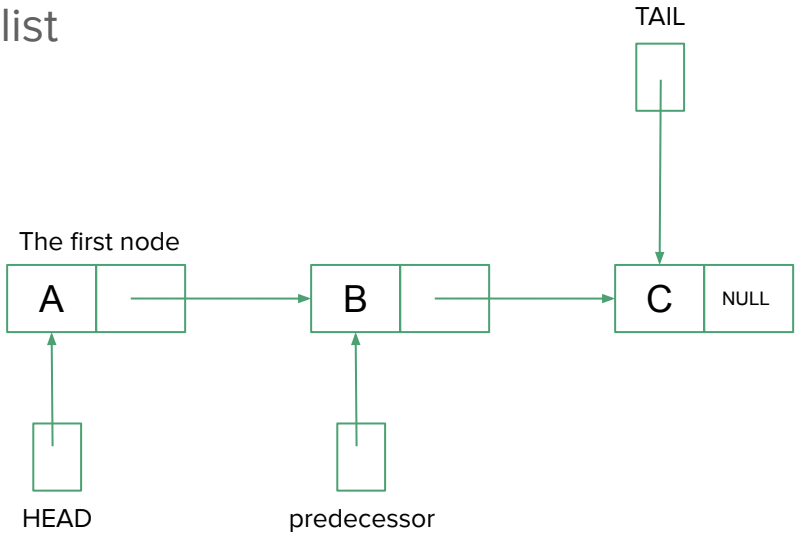
Algorithms

Algorithm: add(data, predecessor)

Input: A linked list, list(HEAD, TAIL), the data to be added, and the predecessor node

Output: The updated list with data added to the list

Steps:

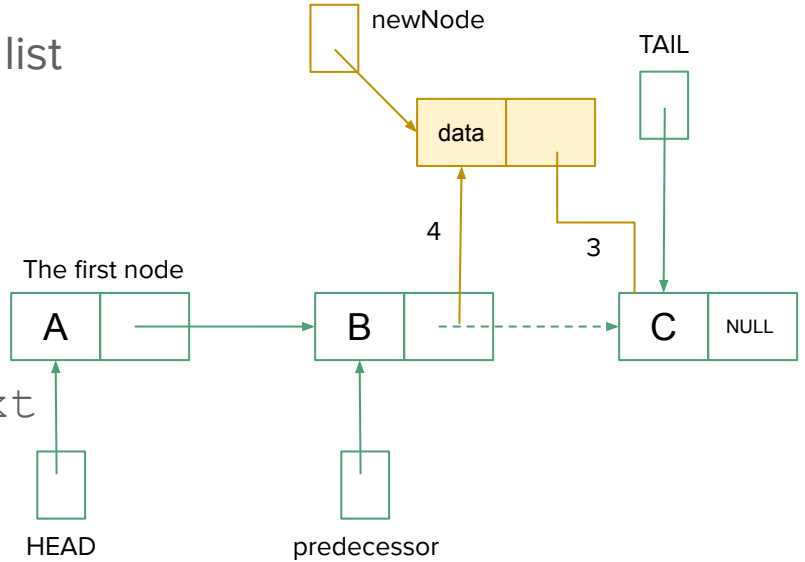


Algorithms

Algorithm: add(data, predecessor)
Input: A linked list, list(HEAD, TAIL), the data to be added, and the predecessor node
Output: The updated list with data added to the list

Steps:

1. Create a new node, `newNode`
2. `newNode->info = data`
3. `newNode->next = predecessor->next`
4. `predecessor->next = newNode`



Algorithms

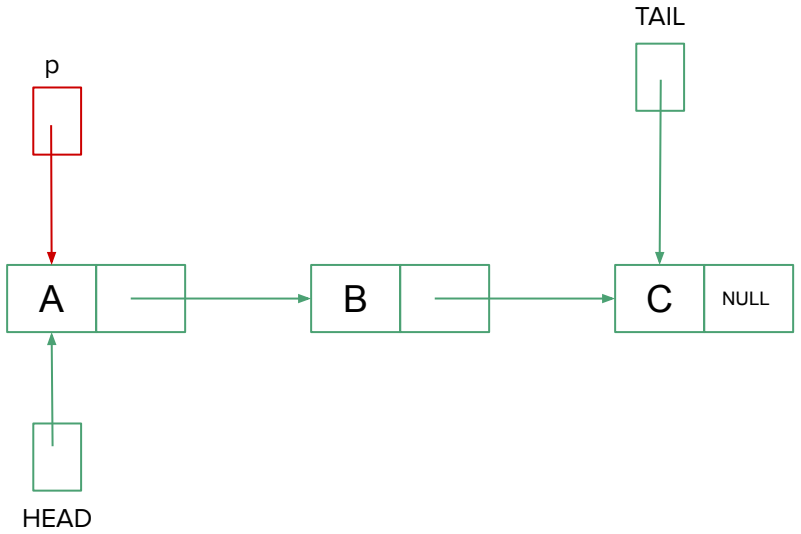
Algorithm: traverse

Input: A linked list, list(HEAD, TAIL)

Output: All list elements are displayed

Steps:

1. Set `p = HEAD`
2. while (`p != NULL`)
   a. Print `p->info`
   b. `p = p->next`
3. endwhile



Algorithms

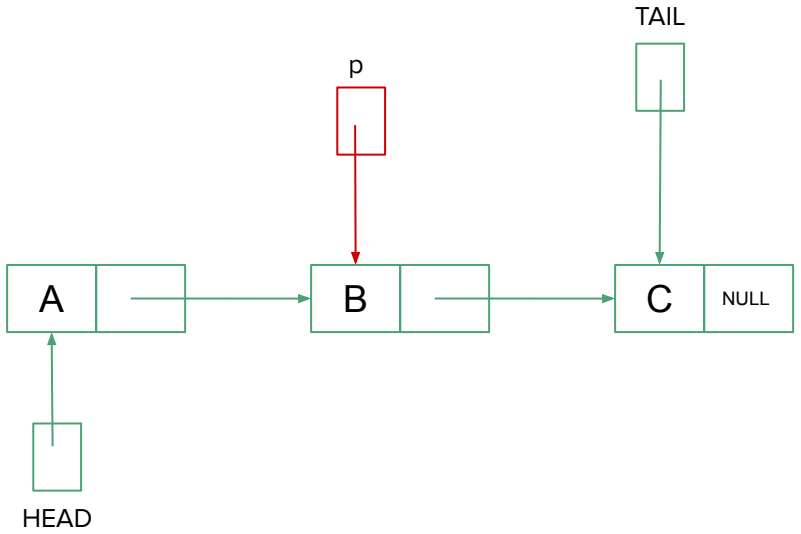
Algorithm: traverse

Input: A linked list, list(HEAD, TAIL)

Output: All list elements are displayed

Steps:

1. Set `p = HEAD`
2. while (`p != NULL`)
   a. Print `p->info`
   b. `p = p->next`
3. endwhile



Algorithms

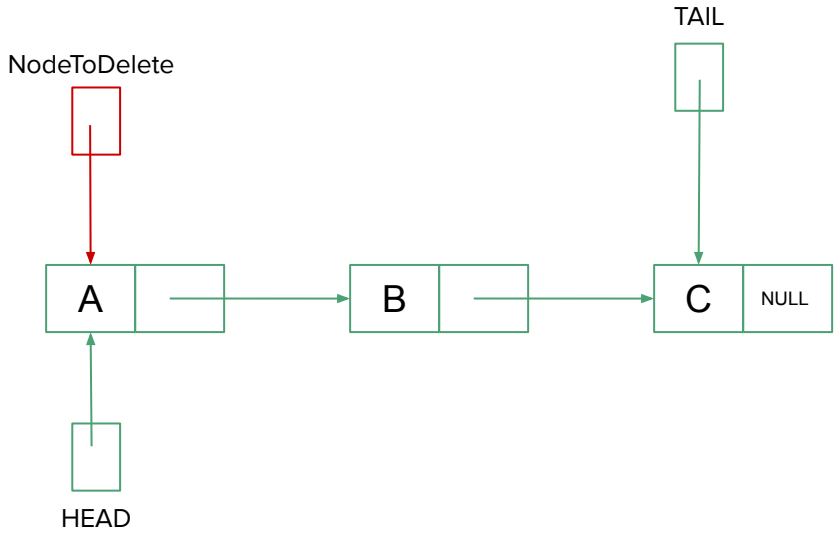
Algorithm: removeFromHead

Input: A linked list, list(HEAD, TAIL)

Output: The updated list with the first node removed

Steps:

1. Set `NodeToDelete = HEAD`
2. `HEAD = NodeToDelete->next`
3. Delete `NodeToDelete`



Algorithms

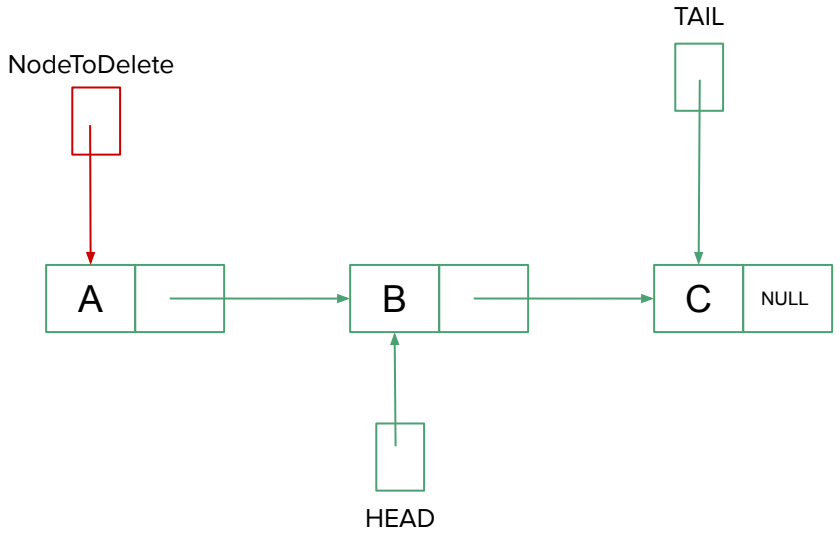
Algorithm: removeFromHead

Input: A linked list, list(HEAD, TAIL)

Output: The updated list with the first node removed

Steps:

1. Set `NodeToDelete = HEAD`
2. `HEAD = NodeToDelete->next`
3. Delete `NodeToDelete`



Algorithms

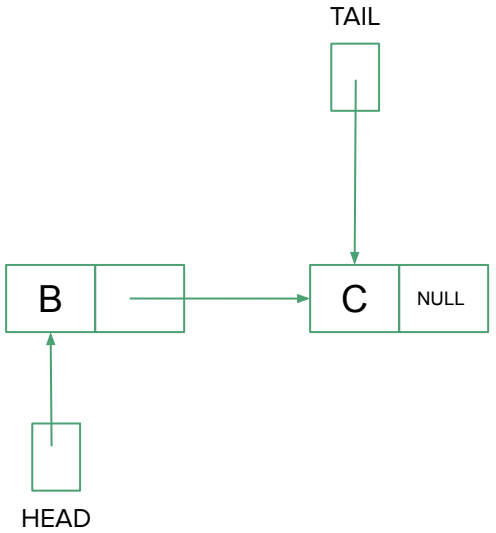
Algorithm: removeFromHead

Input: A linked list, list(HEAD, TAIL)

Output: The updated list with the first node removed

Steps:

1. If the list is not empty
    a. Set `NodeToDelete = HEAD`
    b. `HEAD = NodeToDelete->next`
    c. Delete `NodeToDelete`
2. endif



Algorithms

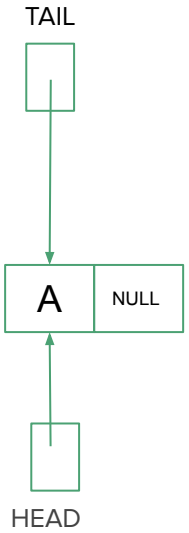
Algorithm: removeFromHead

Input: A linked list, list(HEAD, TAIL)

Output: The updated list with the first node removed

Steps:

1. If the list is not empty
   a. Set `NodeToDelete = HEAD`
   b. `HEAD = NodeToDelete->next`
   c. Delete `NodeToDelete`
   d. If `HEAD == NULL` // If the list is empty now
      i. `TAIL = NULL`
   e. endif
2. endif



Algorithms

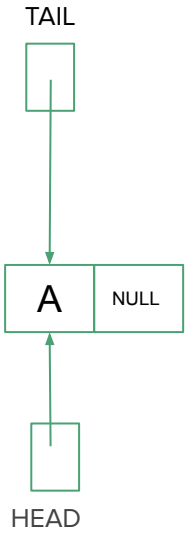
Algorithm: removeFromTail

Input: A linked list, list(HEAD, TAIL)

Output: The updated list with the last node removed

Steps:

1. If the list is not empty
    a. Set NodeToDelete = TAIL
    b. if (HEAD == TAIL)
        i. HEAD = TAIL = NULL



Algorithms

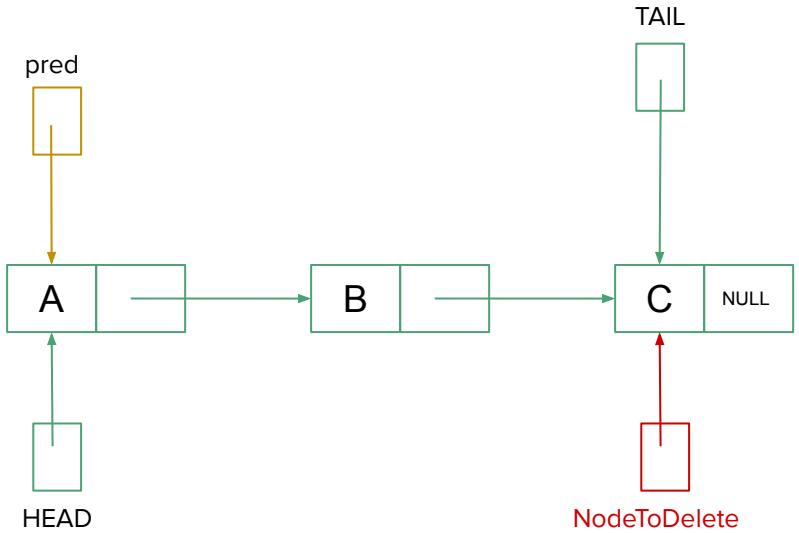
Algorithm: removeFromTail

Input: A linked list, list(HEAD, TAIL)

Output: The updated list with the last node removed

Steps:

1. If the list is not empty
    a. Set NodeToDelete = TAIL
    b. if (HEAD == TAIL)
        i. HEAD = TAIL = NULL
    c. Else
        i. Set pred = HEAD
        ii. while (pred->next != TAIL)
            1. pred = pred->next
        iii. Endwhile
        iv. TAIL = pred
        v. pred->next = NULL
    d. Endif
    e. Delete NodeToDelete
2. Endif



Algorithms

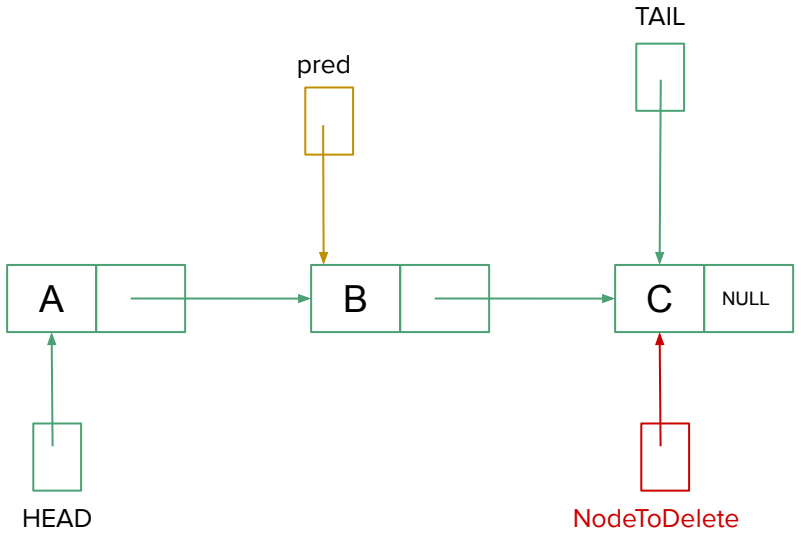
Algorithm: removeFromTail

Input: A linked list, list(HEAD, TAIL)

Output: The updated list with the last node removed

Steps:

1. If the list is not empty
    a. Set NodeToDelete = TAIL
    b. if (HEAD == TAIL)
        i. HEAD = TAIL = NULL
    c. Else
        i. Set pred = HEAD
        ii. while (pred->next != TAIL)
            1. pred = pred->next
        iii. Endwhile
        iv. TAIL = pred
        v. pred->next = NULL
    d. Endif
    e. Delete NodeToDelete
2. Endif



Algorithms

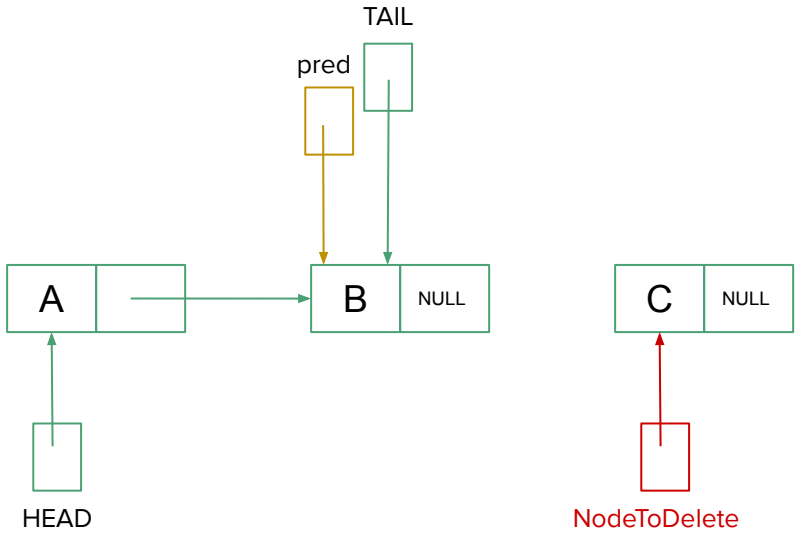
Algorithm: removeFromTail

Input: A linked list, list(HEAD, TAIL)

Output: The updated list with the last node removed

Steps:

1. If the list is not empty
    a. Set NodeToDelete = TAIL
    b. if (HEAD == TAIL)
        i. HEAD = TAIL = NULL
    c. Else
        i. Set pred = HEAD
        ii. while (pred->next != TAIL)
            1. pred = pred->next
        iii. Endwhile
        iv. TAIL = pred
        v. pred->next = NULL
    d. Endif
    e. Delete NodeToDelete
2. Endif



Algorithms

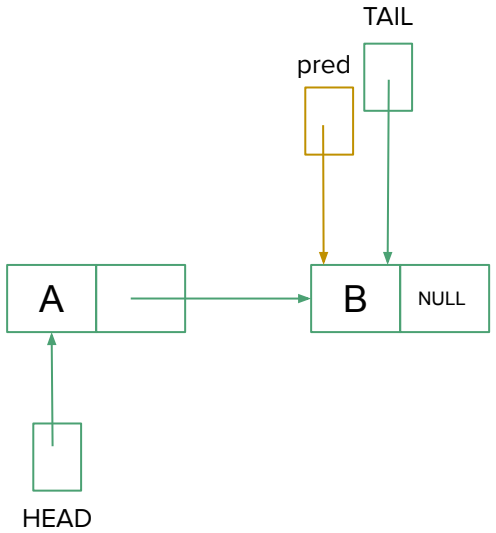
Algorithm: removeFromTail

Input: A linked list, list(HEAD, TAIL)

Output: The updated list with the last node removed

Steps:

1. If the list is not empty
    a. Set NodeToDelete = TAIL
    b. if (HEAD == TAIL)
        i. HEAD = TAIL = NULL
    c. Else
        i. Set pred = HEAD
        ii. while (pred->next != TAIL)
            1. pred = pred->next
        iii. Endwhile
        iv. TAIL = pred
        v. pred->next = NULL
    d. Endif
    e. Delete NodeToDelete
2. Endif



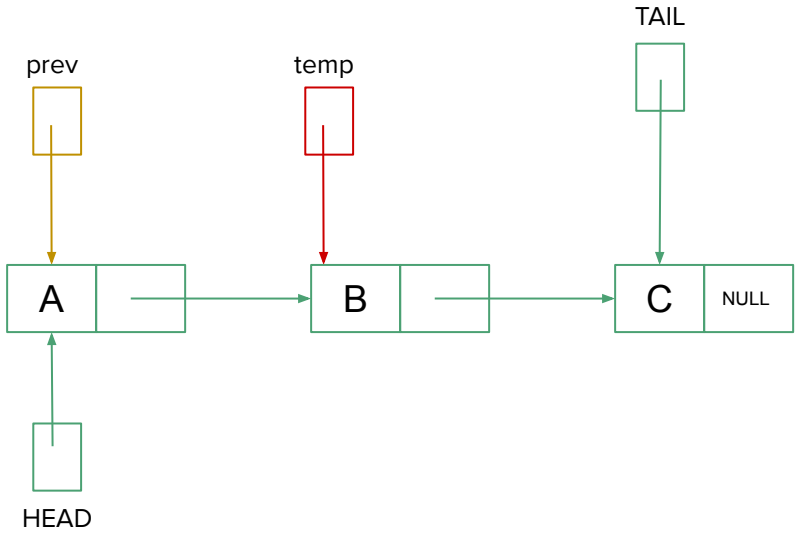
Algorithms

Algorithm: remove(data)
Input: A linked list, list(HEAD, TAIL), and the data to be removed
Output: The updated list with the node containing the given data removed
Steps:

1. If the list is not empty
    1.1. If HEAD->info == data
        1.1.1. removeFromHEAD()
    1.2. Else
        1.2.1. Set temp = HEAD->next
        1.2.2. Set prev = HEAD
        1.2.3.
        1.2.4.

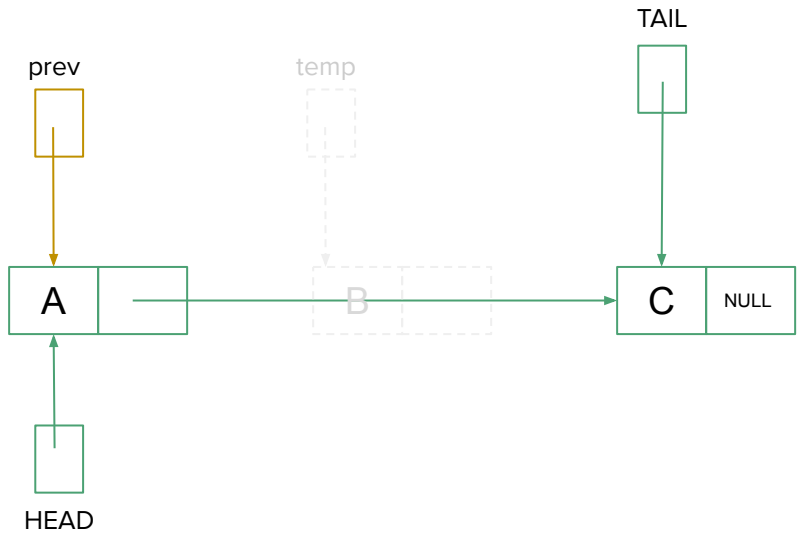
Algorithm: remove(data) (Contd.)

1.1.3. while (temp != NULL)
  1.1.3.1. If (temp->info == data)
    1.1.3.1.1. break
  1.1.3.2. else
    1.1.3.2.1. prev = prev->next
    1.1.3.2.2. temp = temp->next
  1.1.3.3. endif
1.1.4. endwhile



Algorithm: remove(data) (Contd.)

1.1.3. while (temp != NULL)
  1.1.3.1. If (temp->info == data)
    1.1.3.1.1. break
  1.1.3.2. else
    1.1.3.2.1. prev = prev->next
    1.1.3.2.2. temp = temp->next
  1.1.3.3. endif
1.1.4. endwhile
1.1.5. If (temp != NULL)
  1.1.5.1. prev->next = temp->next
  1.1.5.2. Remove temp
  1.1.5.3. If prev->next == NULL
    1.1.5.3.1. TAIL = prev
  1.1.5.4. endif
1.1.6. endif
1.3. endif



Algorithms

Algorithm: retrieve(data, outputPtr)
Input: A linked list, list(HEAD, TAIL), data to search
Output: Pointer to the node containing the requested data
Steps:

1. Set p = HEAD
2. while (p != NULL and p->info != data)
    a. p = p->next
3. Endwhile
4. If (p == NULL)
    a. return false
5. else
    a. outputPtr = p
    b. return true
6. endif