Data Structures and Algorithms

Lecture 2: Linked Lists

Department of Computer Science & Technology
United International College

Outline

- Abstract Data Type (ADT)
- List ADT
- Linked lists

Abstract Data Type

- Data type = Data + Operation
  - Example 1: integer
    - Data: a whole number
    - Operations: +, -, x, /, …
  - Example 2: string
    - Data: an array of characters
    - Operations: strlen, strcpy, strcat, strcmp, …
- Can this be generalized?
  - Abstract Data Type (ADT)
  - Encapsulation

Encapsulation - What

- Users of Data
  - do not touch data directly
  - operates on data by calling the methods
  - do not know how the methods are implemented

Encapsulation - Why

- Modular: one module for one ADT
  - Implementation of the ADT is separate from its use
  - Allows parallel development
  - Easier to debug
- Code for the ADT can be reused in different applications
- Information hiding
  - Protect data from unwanted operations
  - implementation details can be changed without affecting user programs
- Allow rapid prototyping
  - Prototype with simple ADT implementations, then tune them later when necessary

Encapsulation - How

- In OOP Languages:
  - ADT: Class
  - Data: member variables
  - Methods: member functions
- In C:
  - Data: variables (usually of a struct data type)
  - Methods: functions
  - *Information hiding is not supported in C*

The List ADT - Data

- A sequence of zero or more elements

$$A_1, A_2, A_3, \ldots A_N$$

  - N: length of the list
  - A_1: first element
  - A_N: last element
  - A_i: element at position i
  - If N=0, then it is an empty list
- Linearly ordered
  - A_i precedes A_{i+1}
  - A_i follows A_{i-1}
- The elements can of any data type but we use double for discussion

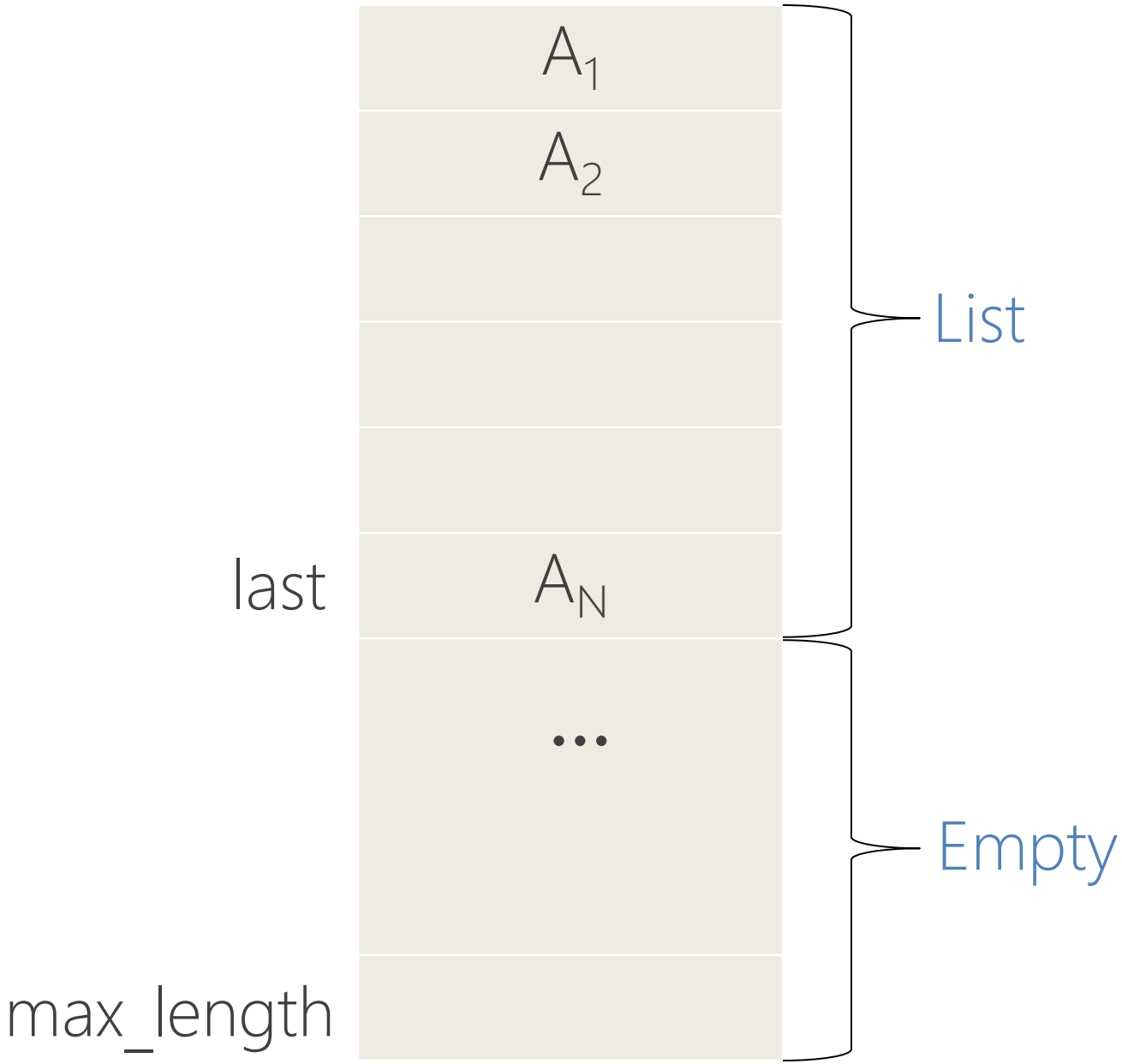
The List ADT - Operations

- makeEmpty: create an empty list
- insert: insert an object to a list
  - insert(x,3) → 34, 12, 52, x, 16, 12
- remove: delete an element from the list
  - remove(52) → 34, 12, x, 16, 12
- find: locate the position of an object in a list
  - list: 34,12, 52, 16, 12
  - find(52) → 3
- findKth: retrieve the element at a certain position
- printList: print the list

Implementation of an ADT

- Define data using data types
- Define operation using functions
- Two standard implementations for the list ADT
  - Array-based
  - Linked list

Array Implementation

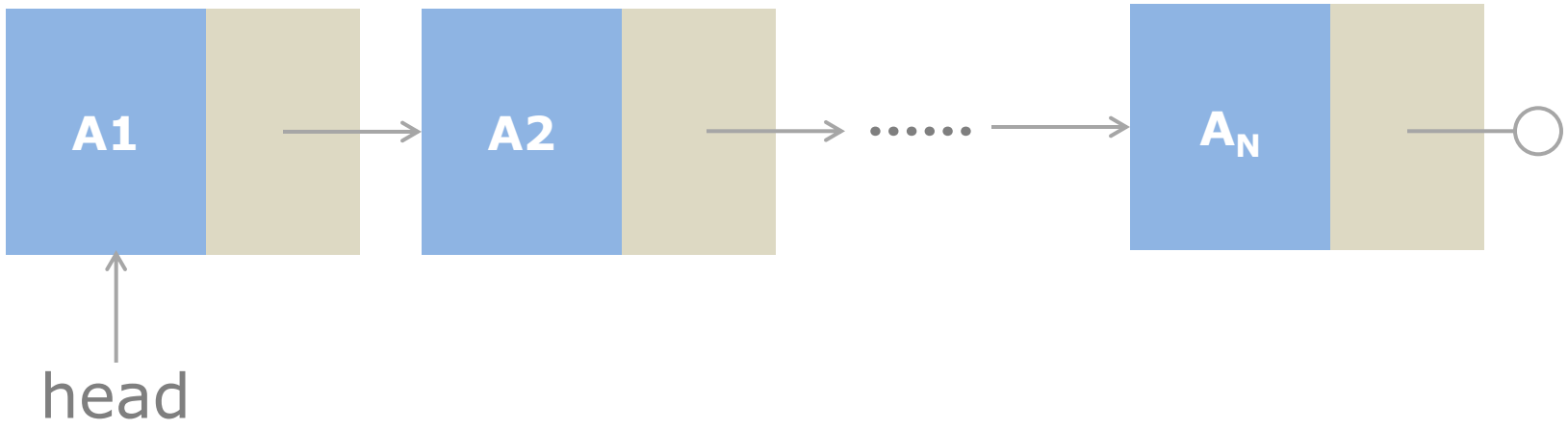


Discussion

- Are these operations suitable for the array implementation?
  - insert
  - remove
  - find
  - findKth
- Any additional pros and cons?

Pointer Implementation (Linked List)

- Ensure that the list is not stored contiguously
  - A node stores one element
  - The address of a node is stored in its previous node
  - The address of the first node must be stored



Discussion

- Are these operations suitable for the linked list?
  - insert
  - remove
  - find
  - findKth
- Any additional pros and cons?

A Complete list of Comparison

Topic		Array	Linked List
Efficiency	insert		
	remove		
	find		
	findKth		
space			

Linked List Implementation

- Data
  - Nodes

```
typedef struct node{
        double data;
        struct node* next;
} Node;
```

  - A pointer to the first node

```
Node* head;
```

Linked List Implementation

- Methods

```
bool IsEmpty(Node* head);
Node* InsertNode(Node** phead, int index, double x);
int FindNode(Node* head, double x);
int DeleteNode(Node** phead, double x);
void DisplayList(Node* head);
Void DestroyList(Node* head);
```

Methods

- bool IsEmpty(Node* head)
  - returns *true* if the list is empty and *false* otherwise
- Nodes* InsertNode(Node ** phead, int index, double x)
  - insert a new node after position *index*
  - position of nodes starts from 1
  - insert a new node as the head if *index=0*
  - returns a pointer to the new node if insertion is successful and *NULL* otherwise
- int FindNode(Node* head, double x)
  - returns the position of the first node whose *data=x*
  - returns 0 if no such node exists

Methods

- int DeleteNode(Node** phead, double x)
  - deletes the first node whose *data=x*
  - returns the position of the deleted node
  - returns 0 if no such node exists
- void DisplayList(Node* head)
  - prints all the nodes in the list
- void DestroyList(Node* head)
  - deletes all the nodes in the list
  - frees the memory allocated to the nodes

Insert

- Node* InsertNode(Node** phead, int index, double x)
  - insert a new node after position *index*
  - position of nodes starts from 1
  - insert a new node as the head if *index=0*
  - returns a pointer to the new node if insertion is successful and *null* otherwise
- Why Node **phead?

Insert

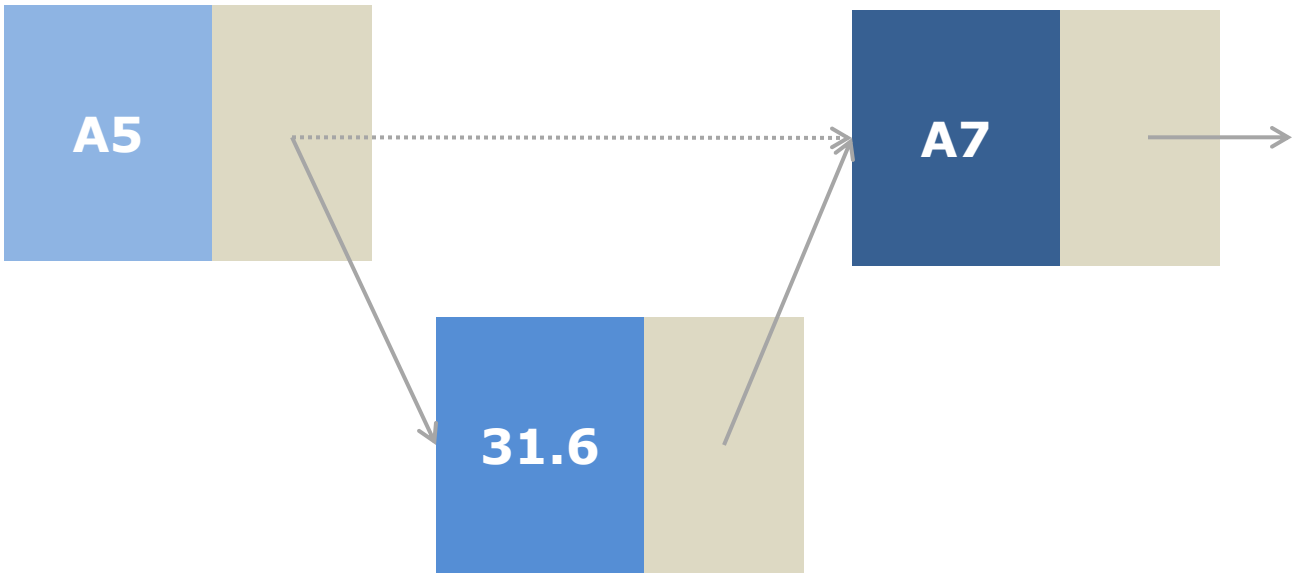
1. Locate the element at position *index*
2. Allocate memory for the new node
3. Point the new node to its successor
4. Point the new node's predecessor to the new node

InsertNode(&head, 5, 31.6)

Insert

1. Locate the element at position *index*
2. Allocate memory for the new node
3. Point the new node to its successor
4. Point the new node's predecessor to the new node

InsertNode(&head, 5, 31.6)



Insert

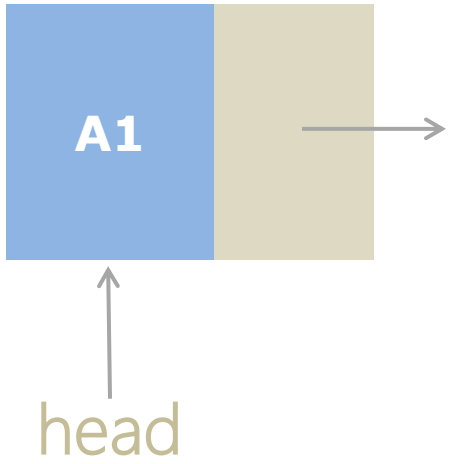
- Possible cases of `InsertNode`
  1. Insert into an empty list
  2. Insert in front
  3. Insert at back
  4. Insert in middle
- But, in fact, only need to handle two cases
  - Insert as the first node (Case 1 and Case 2)
  - Insert in the middle or at the end of the list (Case 3 and Case 4)

Two Cases for Insert

- Insert as the first node
  - handles the *next* pointer of one node
  - updates the *head* pointer
- Insert in the middle or at the end
  - handles the *next* pointer of two nodes

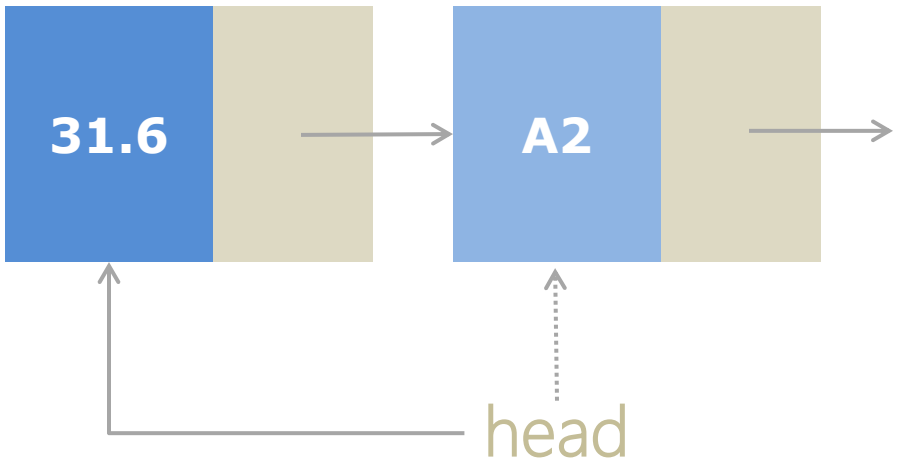
Insert as the First Node

InsertNode(&head, 0, 31.6)



Insert as the First Node

InsertNode(&head, 0, 31.6)



Code for Insert

```
#include <stdlib.h>
Node* InsertNode(Node** phead, int index, double x) {
	if (index < 0) return 0;

	int currIndex = 1;
	Node* currNode = *phead;
	while (currNode && index > currIndex) {
		currNode = currNode->next;
		currIndex ++;
	}
	if (index > 0 && currNode == 0) return 0;

	Node* newNode = (Node*)malloc(sizeof(Node));
	newNode->data = x;
	if (index == 0) {
		newNode->next = *phead;
		*phead = newNode;
	}
	else {
		newNode->next = currNode->next;
		currNode->next = newNode;
	}
	return newNode;
}
```

```
#include <stdlib.h>
Node* InsertNode(Node** phead, int index, double x) {
        if (index < 0) return 0;

        int currIndex = 1;
        Node* currNode = *phead;
        while (currNode && index > currIndex) {
                currNode = currNode->next;
                currIndex ++;
        }
        if (index > 0 && currNode == 0) return 0;

        Node* newNode = (Node*)malloc(sizeof(Node));
        newNode->data = x;
        if (index == 0) {
                newNode->next = *phead;
                *phead = newNode;
        }
        else {
                newNode->next = currNode->next;
                currNode->next = newNode;
        }
        return newNode;
}
```

Try to locate the node at position *index*. If it does not exist, return *null*.

```
#include <stdlib.h>
Node* InsertNode(Node** phead, int index, double x) {
	if (index < 0) return 0;

	int currIndex = 1;
	Node* currNode = *phead;
	while (currNode && index > currIndex) {
		currNode = currNode->next;
		currIndex ++;
	}
	if (index > 0 && currNode == 0) return 0;

	Node* newNode = (Node*)malloc(sizeof(Node));
	newNode->data = x;
	if (index == 0) {
		newNode->next = *phead;
		*phead = newNode;
	}
	else {
		newNode->next = currNode->next;
		currNode->next = newNode;
	}
	return newNode;
}
```

Create a new Node.

```
#include <stdlib.h>
Node* InsertNode(Node** phead, int index, double x) {
	if (index < 0) return 0;

	int currIndex = 1;
	Node* currNode = *phead;
	while (currNode && index > currIndex) {
		currNode = currNode->next;
		currIndex ++;
	}
	if (index > 0 && currNode == 0) return 0;

	Node* newNode = (Node*)malloc(sizeof(Node));
	newNode->data = x;
	if (index == 0) {
		newNode->next = *phead;
		*phead = newNode;
	}
	else {
		newNode->next = currNode->next;
		currNode->next = newNode;
	}
	return newNode;
}
```

Insert as the new head.

```
#include <stdlib.h>
Node* InsertNode(Node** phead, int index, double x) {
	if (index < 0) return 0;

	int currIndex = 1;
	Node* currNode = *phead;
	while (currNode && index > currIndex) {
		currNode = currNode->next;
		currIndex ++;
	}
	if (index > 0 && currNode == 0) return 0;

	Node* newNode = (Node*)malloc(sizeof(Node));
	newNode->data = x;
	if (index == 0) {
		newNode->next = *phead;
		*phead = newNode;
	}
	else {
		newNode->next = currNode->next;
		currNode->next = newNode;
	}
	return newNode;
}
```

Insert after currNode.

Find

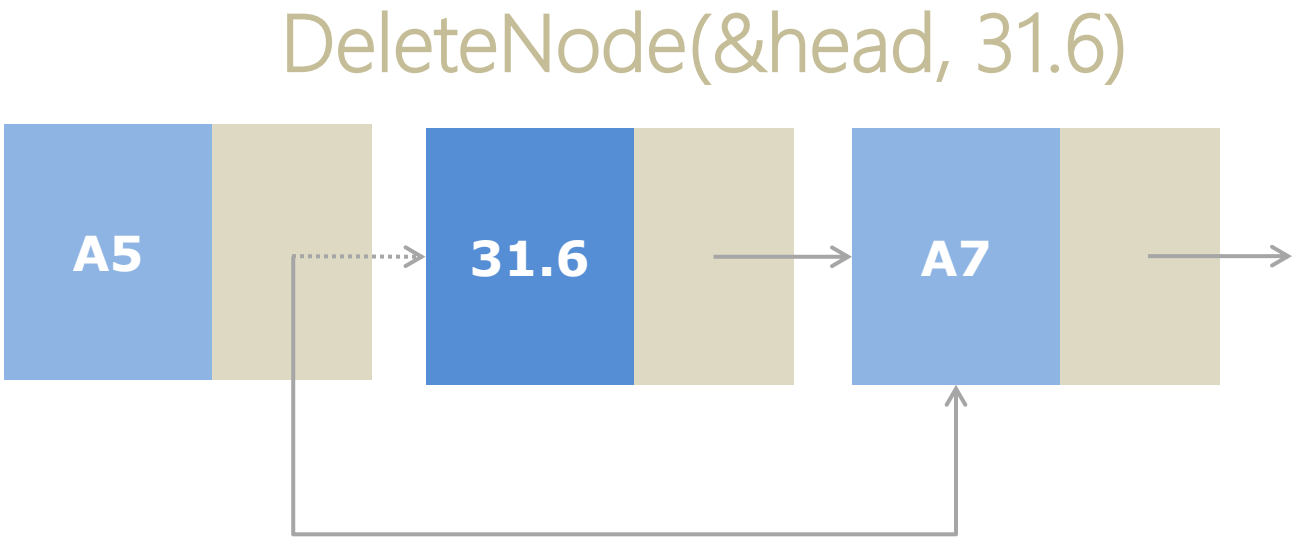
- int FindNode(Node* head, double x)
  - returns the position of the first node whose *data=x*
  - returns 0 if no such node exists
- Steps
  1. Search for a node with the value equal to `x` in the list.
  2. If such a node is found, return its position. Otherwise, return 0.

Delete

- int DeleteNode(Node** phead, double x)
  - deletes a node whose *data=x*
  - returns the position of the deleted node
  - returns 0 if no such node exists
- Steps
  1. Find the desirable node (similar to *FindNode*)
  2. In addition, record the node's predecessor
  3. Free the memory occupied by the found node
  4. Set the pointers

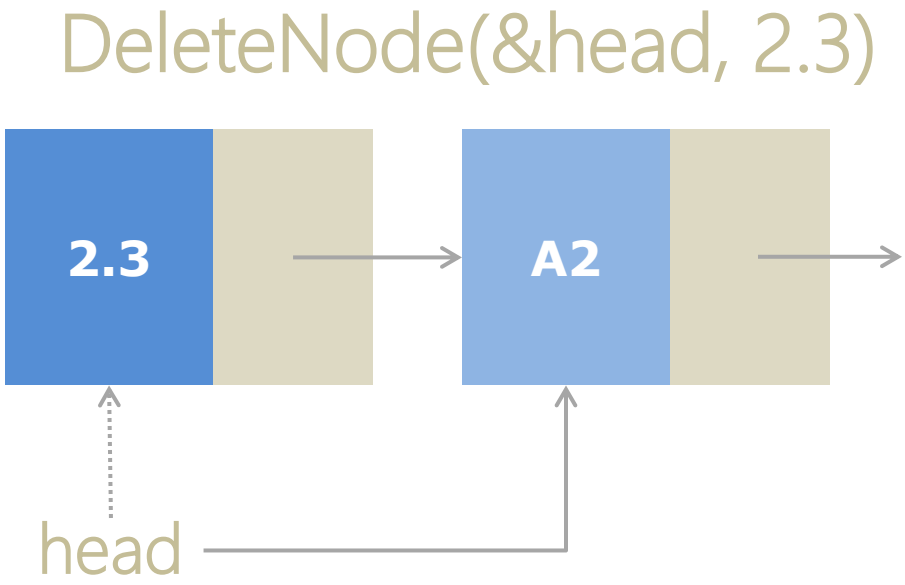
Delete

- Deleting a middle or an end node



Delete

- Deleting the head



Destory

- void DestroyList(Node* head)
  - deletes all the nodes in the list
  - frees the memory allocated to the nodes
- Steps
  - Step through the list and delete each node one by one.
  - Before deleting a node, obtain its next node.

Task

- Given *list.h*, complete *list.cpp* which implements all the functions defined.
- Submit list.cpp to iSpace.