Data Structures and Algorithms

Linked Lists

Stacks

PLSD210(ii)

Array Limitations

- **Arrays**
  - **Simple,**
  - **Fast**

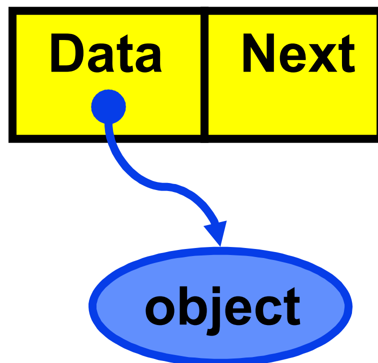
  but
  - **Must specify size at construction time**
  - **Murphy's law**
    - **Construct an array with space for n**
      - **n = twice your estimate of largest collection**
    - **Tomorrow you'll need $n+1$**
  - **More flexible system?**

Linked Lists

- **Flexible space use**
  - **Dynamically allocate space for each element as needed**
  - **Include a pointer to the next item**

←Linked list

  - **Each node of the list contains**
    - **the data item** (an object pointer in our ADT)
    - **a pointer to the next node**



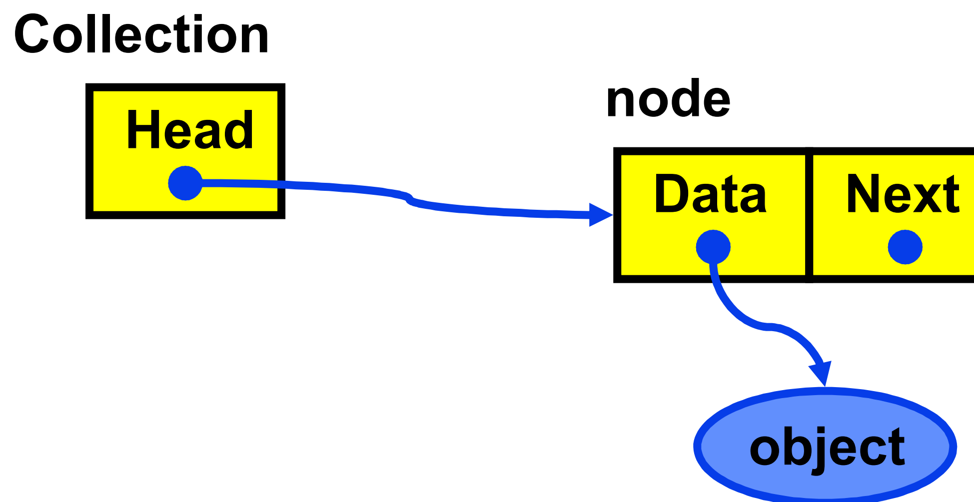
Linked Lists

- **Collection structure has a pointer to the list head**
  - **Initially NULL**

Collection

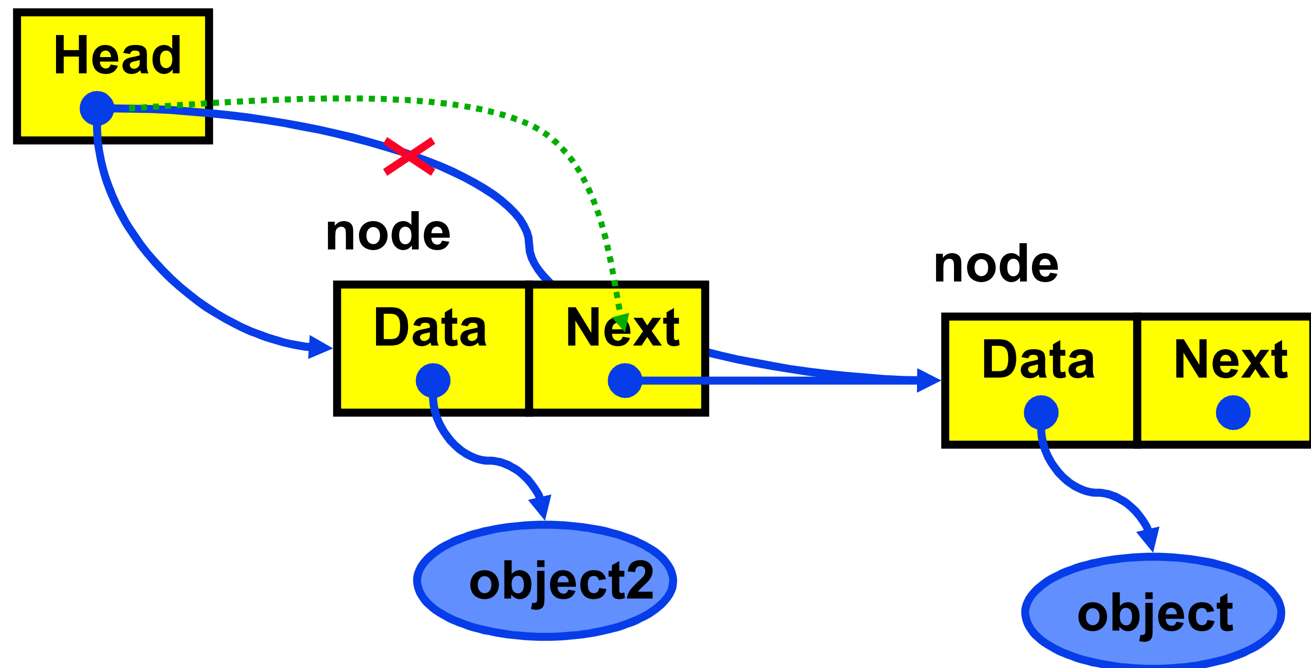
Linked Lists

- **Collection structure has a pointer to the list head**
  - **Initially NULL**
- **Add first item**
  - **Allocate space for node**
  - **Set its data pointer to object**
  - **Set Next to NULL**
  - **Set Head to point to new node**



Linked Lists

- Add second item
  - Allocate space for node
  - Set its data pointer to object
  - Set Next to current Head
  - Set Head to point to new node



Linked Lists - `Add` *implementation*

- **Implementation**

```
struct t_node {
    void *item;
    struct t_node *next;
    } node;
typedef struct t_node *Node;
struct collection {
    Node head;
    ……
    };
int AddToCollection( Collection c, void *item ) {
    Node new = malloc( sizeof( struct t_node ) );
    new->item = item;
    new->next = c->head;
    c->head = new;
    return TRUE;
    }
```

Linked Lists - `Add` *implementation*

- **Implementation**

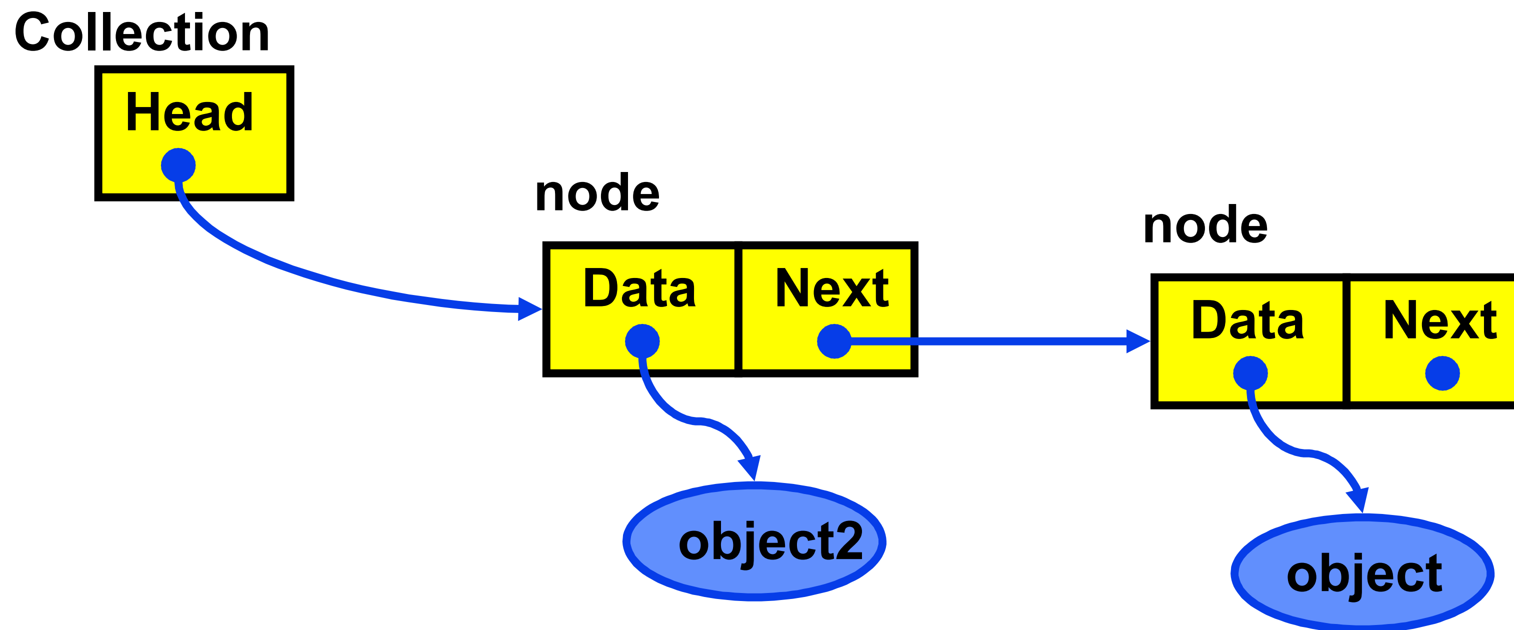
```
struct t_node {
    void *item;
    struct t_node *next;
    } node;
typedef struct t_node *Node;
struct collection {
    Node head;
    ……
    };
int AddToCollection( Collection c, void *item ) {
    Node new = malloc( sizeof( struct t_node ) );
    new->item = item;
    new->next = c->head;
    c->head = new;
    return TRUE;
    }
```

Recursive type definition - C allows it!

Error checking, asserts omitted for clarity!

Linked Lists

- Add time
  - Constant - independent of n
- Search time
  - Worst case - n



Linked Lists - `Find` *implementation*

- **Implementation**

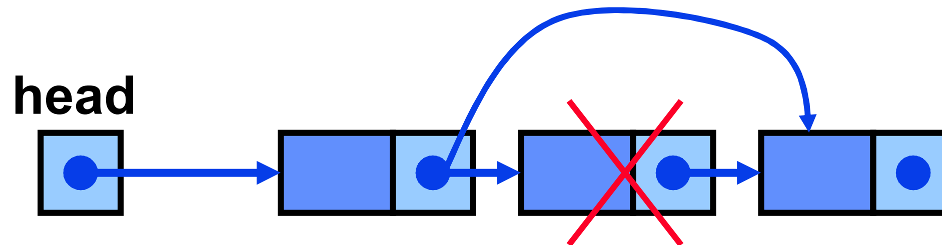
```
void *FindinCollection( Collection c, void *key ) {
    Node n = c->head;
    while ( n != NULL ) {
      if ( KeyCmp( ItemKey( n->item ), key ) == 0 ) {
            return n->item;
      n = n->next;
      }
    return NULL;
    }
```

- *A recursive implementation is also possible!*

Linked Lists - `Delete` *implementation*

- **Implementation**

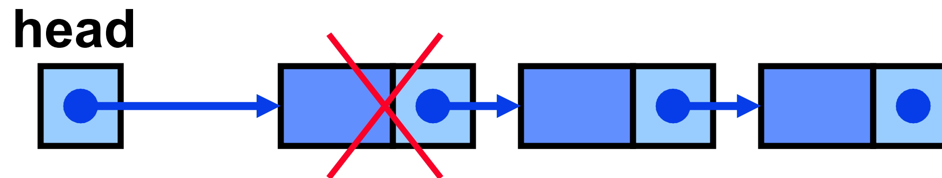
```
void *DeleteFromCollection( Collection c, void *key ) {
    Node n, prev;
    n = prev = c->head;
    while ( n != NULL ) {
      if ( KeyCmp( ItemKey( n->item ), key ) == 0 ) {
            prev->next = n->next;
            return n;
            }
      prev = n;
      n = n->next;
      }
    return NULL;
    }
```



Linked Lists - `Delete` *implementation*

- **Implementation**

```
void *DeleteFromCollection( Collection c, void *key ) {
    Node n, prev;
    n = prev = c->head;
    while ( n != NULL ) {
      if ( KeyCmp( ItemKey( n->item ), key ) == 0 ) {
            prev->next = n->next;
            return n;
            }
      prev = n;
      n = n->next;
      }
    return NULL;
    }
```

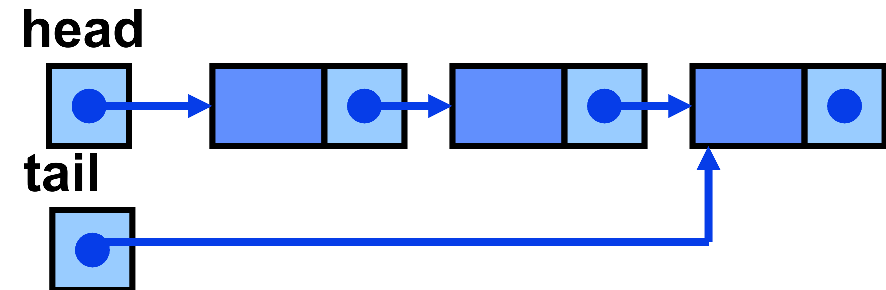


Minor addition needed to allow for deleting this one! An exercise!

Linked Lists - LIFO and FIFO

- **Simplest implementation**
  - **Add to head**

  ← Last-In-First-Out (LIFO) semantics
- **Modifications**
  - **First-In-First-Out (FIFO)**
  - **Keep a tail pointer**




```
struct t_node {
    void *item;
    struct t_node *next;
    } node;
typedef struct t_node *Node;
struct collection {
    Node head, tail;
    };
```

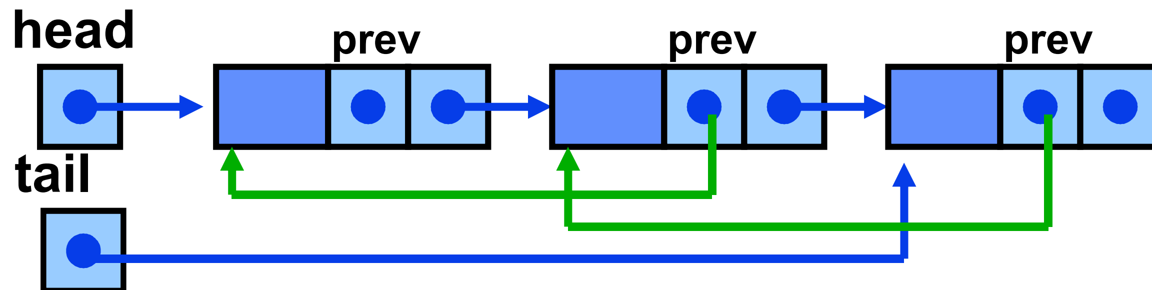
`tail` is set in the `AddToCollection` method if `head == NULL`

Linked Lists - Doubly linked

- Doubly linked lists
  - Can be scanned in both directions

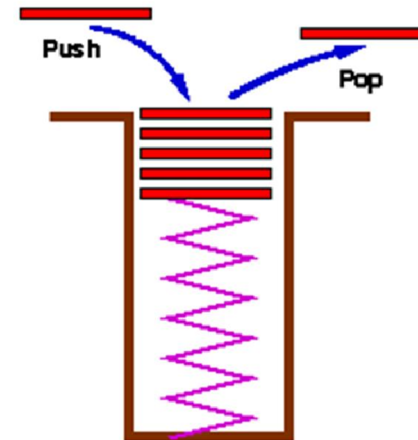
```
struct t_node {
    void *item;
    struct t_node *prev,
                  *next;
    } node;

typedef struct t_node *Node;
struct collection {
    Node head, tail;
    };
```



Stacks

- Stacks are a special form of collection with LIFO semantics
- Two methods
  - `int push( Stack s, void *item );`
    - add item to the top of the stack
  - `void *pop( Stack s );`
    - remove an item from the top of the stack
- Like a plate stacker
- Other methods




```
int IsEmpty( Stack s );
/* Return TRUE if empty */
void *Top( Stack s );
/* Return the item at the top,
    without deleting it */
```

Stacks - Implementation

- Arrays
  - Provide a stack capacity to the constructor
  - Flexibility limited *but* matches many real uses
    - Capacity limited by some constraint
      - Memory in your computer
      - Size of the plate stacker, etc
- `push, pop` methods
  - Variants of `AddToC…, DeleteFromC…`
- Linked list also possible
- Stack:
  - *basically a Collection with special semantics!*

Stacks - Relevance

- **Stacks appear in computer programs**
  - **Key to call / return in functions & procedures**
  - **Stack frame allows recursive calls**
  - **Call: push stack frame**
  - **Return: pop stack frame**
- **Stack frame**
  - **Function arguments**
  - **Return address**
  - **Local variables**

Stacks - Implementation

- **Arrays common**
  - Provide a stack capacity to the constructor
  - Flexibility limited but matches many real uses
    - Stack created with limited capacity

```
struct t_node {
    void *item;
    struct t_node *prev,
                  *next;
    } node;

typedef struct t_node *Node;
struct collection {
    Node head, tail;
    };
```

prev is optional!