Linked List - Data Structures

Dr. Animesh Chaturvedi

Assistant Professor: IIIT Dharwad

Young Researcher: Heidelberg Laureate Forum

Postdoc: King's College London & The Alan Turing Institute

PhD: IIT Indore MTech: IIITDM Jabalpur

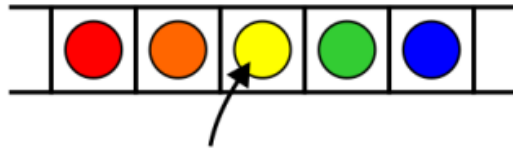# List

- Properties
  - Ordered list of items…precedes, succeeds; first, last
  - Index for each item…lookup or address item by index value
  - Finite Length for the list…can be empty, size can vary
  - Items of same type present in the list
- Operations
  - Create, destroy
  - Lookup by index, item value
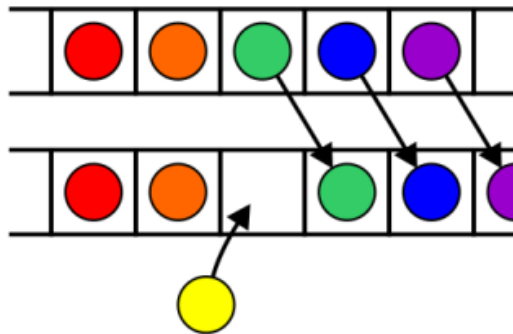 - Find size, if empty
  - Add, delete item

List Operations

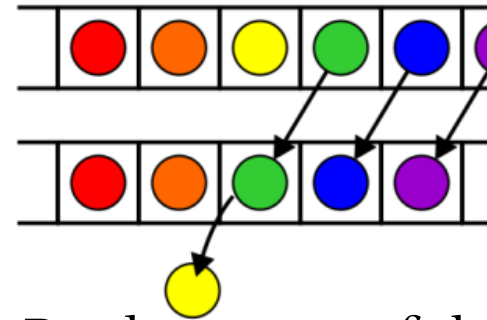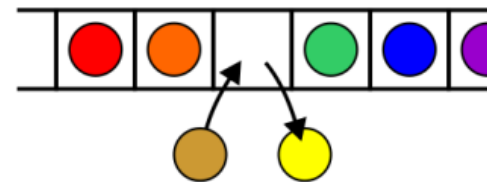- Operations at the kth entry of the list include:

- Access to the object

- Insertion of a new object

Erasing an object

Replacement of the object

List Operations

- Given access to the kth object, gain access to either the previous or next object
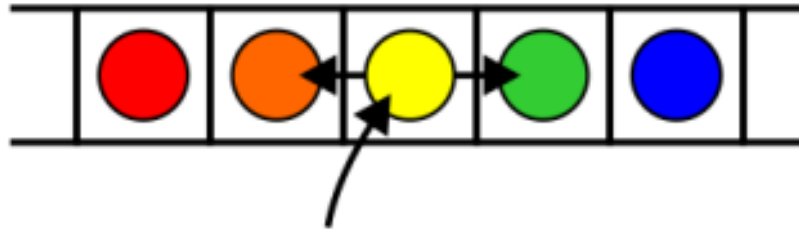
- Given two abstract lists, we may want to
  - Concatenate the two lists
  - Determine if one is a sub-list of the other

Array List

Array List Operations

- List operations
  - Create/Destroy
  - Length
  - Find
  - Insert/Remove
  - Next/Previous
- List properties
  - A_i precedes A_{i+1} for $1 \leq i < n$
  - A_i succeeds A_{i-1} for $1 < i \leq n$
  - Size 0 list is defined to be the **empty list ()**

$$(\ A_1 \ A_2 \ ... \ A_{n-1} \ A_n \ )$$
$$\text{length = n}$$

Array List Operations

- First, create an array list.

- Then, we are able to perform: insert into, access, and erase from the values stored in the array list

Operations on an arbitrary node of the array list,

- Find the number of instances of an integer in the array list:

  int count(int) const;

- Remove all instances of an integer from the array list:

  int erase(int);

Array List Operations - Mapping Function

- Say n is the capacity of the array
- Simple mapping
  - E.g. using array for queues: add from front, remove from back.
  - get(index) = value of data at index
- Reverse mapping
  - E.g. using same array for reverse list.
  - get(index) = value of data at (n - index – 1)
- Hash-Mapping
  - get(index) = value of data at (position(0)+index) % n
  - get(0) = value of data at head

Array List Operations

- void create(initCapacity) or create(size)
  - Create array with initial capacity (optional hint)
- void set(index, item)
  - Use mapping function to set value at position
- void append(item)
  - Insert after current "last" item…use size
- void remove(index) or int erase(int)
  - Remove item at index or Erase item at index and return its data (value as integer)

Array List Operations

- item get(index)
  - Use mapping function to set value at position
  - Student is class with methods getName, get RollNumber and "student" is its object
  - e.g., student.getName(15), student.getRollNumber(15)
- int indexOf(item)
  - Get "first" index of item with given value
  - String student.getRollNumber(15)
- To analyse or check the entire list:
  - Is the linked list empty?

    boolean isEmpty() or bool empty();
  - How many objects are in the list? Counts the number of instances in the list,

    int capacity() or int size()
  - The list is empty when the list_head pointer is set to nullptr

Array List Operations

Increasing capacity

- Start with initial capacity given by user, or default

- When capacity is reached

  - Create array with more capacity, e.g. double it

  - Copy values from old to new array

  - Delete old array space

- Can also be used to shrink space

  - Example: Sparse Matrices

Array List Operations

- Pros and Cons of using Arrays.
- Complexity
  - Storage Complexity: Amount of storage required by the data structure, relative to items stored
  - Computational Complexity: Number of CPU cycles required to perform each data structure operation
  - size(), set(), get(), indexOf()

Linked List

Linked List - Definitions and Terminologies

- A linked list is a sequence of data structures, which are connected together via links.

- Linked list is a sequence of links which contains items.

- Each link contains a connection to another link.

- Important terms to understand the concept of linked list.
  - Data − Each link of a linked list can store a data called an element
  - Next − Each link of a linked list contains a link to the next link called Next
  - List − A list contains the connection link to the first link called Head

- It can be visualized as a chain of nodes, where every node points to the next node.
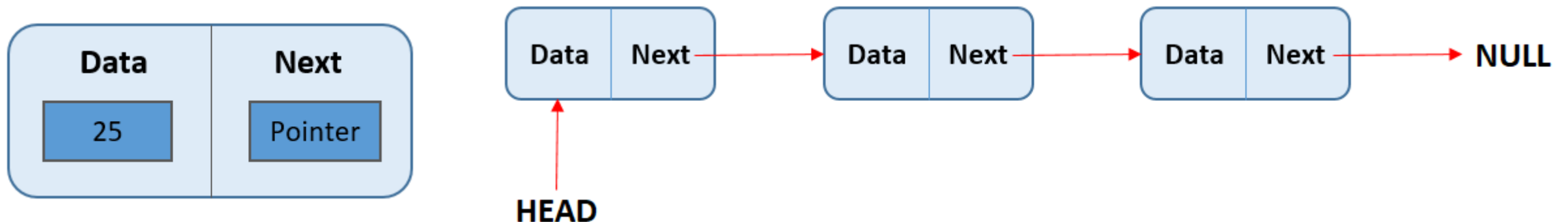
Linked List - Definitions and Terminologies

- A linked list is a data structure where each object is stored in a node

- As well as storing data, the node must also contains a reference/pointer to the node containing the next item of data.

- We must dynamically create the nodes in a linked list

- Thus, because new returns a pointer, the logical manner in which to track a linked lists is through a pointer

- A Node class must store the data and a reference to the next node (also a pointer)

Linked List in C

One or more of its components is a pointer to itself.

```
typedef  struct list {
        char data;
        list *link;
        }
```

```
list item1, item2, item3;
item1.data='a';
item2.data='b';
item3.data='c';
item1.link=item2.link=item3.link=NULL;
```

Construct a list with three nodes
item1.link=&item2;
item2.link=&item3;
malloc: obtain a node
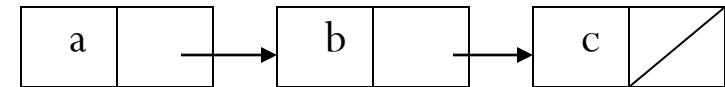
Linked List in C

struct node {

        int data;

        struct node *next;

   };

Data	Next

- Linked list contains a link element called Head
- Each link carries a data field(s) and a link field called next
- Each link is linked with its next link using its next link
- Last link carries a link as null to mark the end of the list

Linked List Representation

- Problem with array:
  - Pre-defined capacity, under usage, cost to move items when full
- Solution:
  - Grow backing data structure dynamically when we add or remove node
    - Only use as much memory as required
- Linked lists use pointers to contiguous chain items
  - Node structure contains item and pointer to next node in List
- Add or remove nodes when setting or getting items

Linked List Operations

- First, create a linked list.

- Then, we are able to perform: insert into, access, and erase from the values stored in the linked list

- Operations on the first node of the linked list

- Adding, retrieving, or removing the value at the front of the linked list

    void push_front(int);

    int front() const;

    void pop_front();

- To access the head of the linked list

    Node *begin() const;

Node Class in C++

The node must store data and a pointer:

```cpp
class Node {
    public: Node( int = 0, Node * = nullptr );


        int value() const;
        Node *next() const;


    private:
            int node_value;
            Node *next_node;
    };
```

Node Constructor in C++

```cpp
class Node {
    public: Node( int = 0, Node * = nullptr );
    int value() const;
    Node *next() const;
    private:
        int node_value;
        Node *next_node;
};
```

The constructor assigns the two member variables based on the arguments

```cpp
List::Node::Node( int e, Node *n ):
    node_value( e )
    next_node( n ) {// empty constructor}
```

The default values are given in the class definition:

Linked List Iterators in C++

General method of examining collections

List<Object> *list;

Object x;

…

ListItr<Object> *i = list->first();

while (i->hasNext()) {

       x = i->next();

}

Linked List Operations

- Iteration operations:
  - ListItr<Object> first()
  - ListItr<Object> kth(int)
  - ListItr<Object> last()
- Main operations:
  - ListItr<Object> find(Object)
  - void insert(Object, listItr<Object>)
  - void remove(ListItr<Object>)
  - bool isEmpty()

Linked List Operations in OOP (C++ or Java)

```
class LinkedList {
    Node* head;

    class Node* {
        int item
        Node* next
    }

    int size() {…}
    append() {…}
    get() {…}
    set() {…}
    remove {…}
}
```

Node* head address
e.g. 0x37

| 6 | Φ |
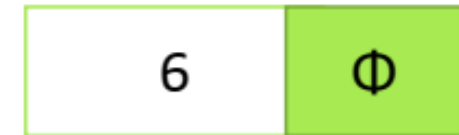
item   Node* next
        e.g. null

Node* head address

| 6 | 0x54 |
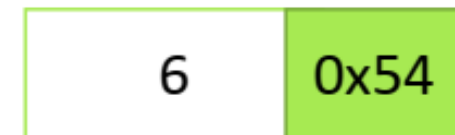
item   Node* next
        e.g. 0x54

Linked List Operations

head=null

Add item 6

0x37

head=0x37 | 6 | Φ

Add item 4

0x37   0x54

head=0x37 | 6 | 0x54 → 4 | Φ

Add items 8, 2

0x37   0x54   0x7A   0xF1

head=0x37 | 6 | 0x54 → 4 | 0x7A → 8 | 0xF1 → 2 | Φ

Remove 3

0x37   0x54   0xF1

head=0x37 | 6 | 0x54 → 4 | 0xF1 → 2 | Φ

Remove 1

0x54   0xF1

head=0x54 | 4 | 0xF1 → 2 | Φ
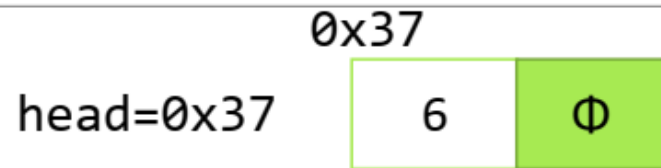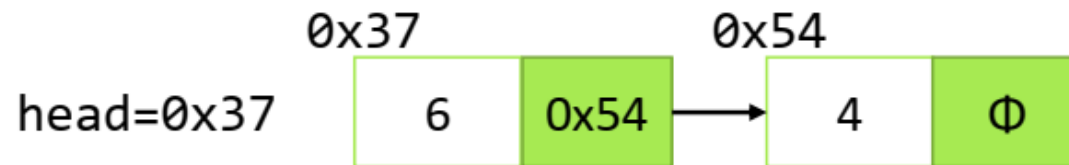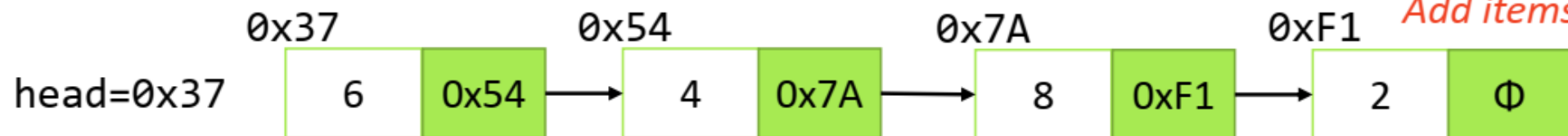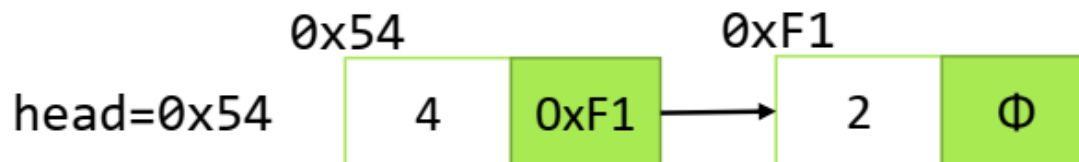
Types of Linked lists

- Depending on the way in which the links are used to maintain adjacency, several different types of linked lists are possible.

- Linear singly-linked list (or simply linear list)
  - One we have discussed so far.
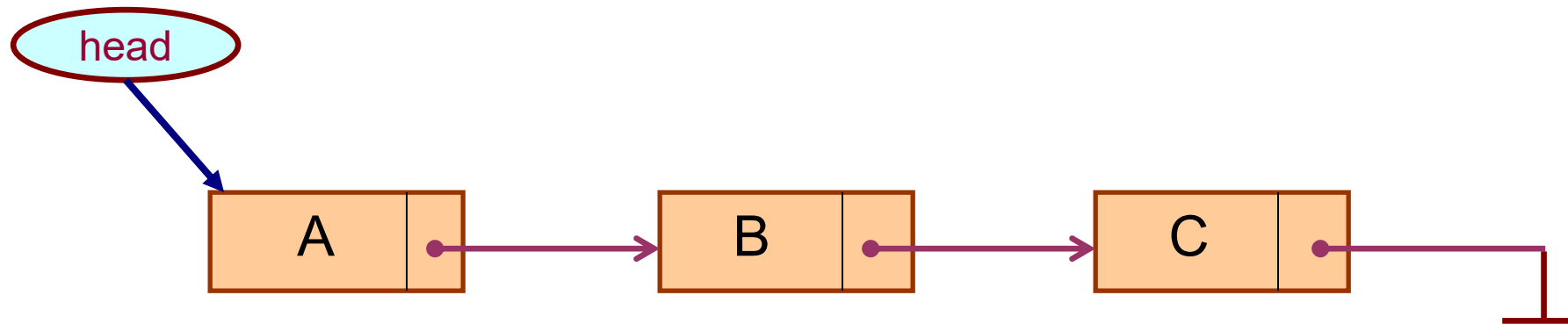
Types of Linked lists

- Doubly linked list
  - Pointers exist between adjacent nodes in both directions.
  - The list can be traversed either forward or backward.
  - Usually two pointers are maintained to keep track of the list, *head* and *tail*.
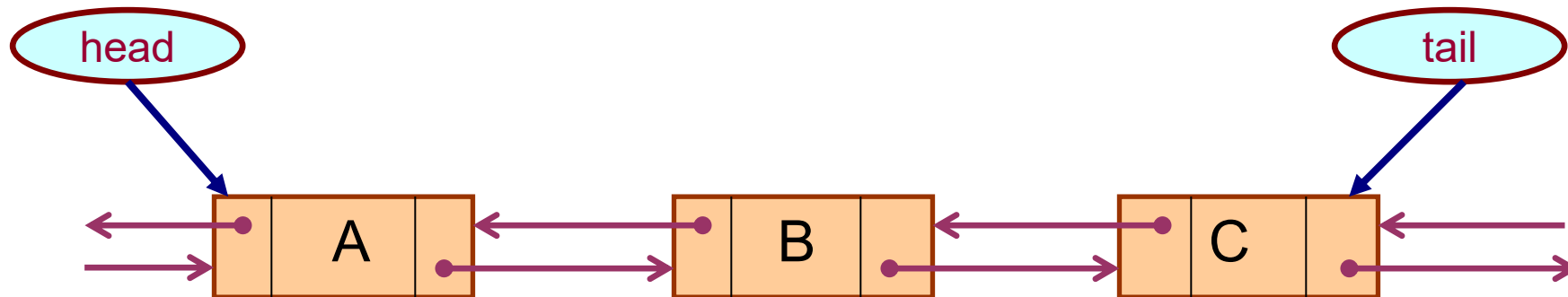
Types of Linked lists

- Circular linked list
  - The pointer from the last element in the list points back to the first element.
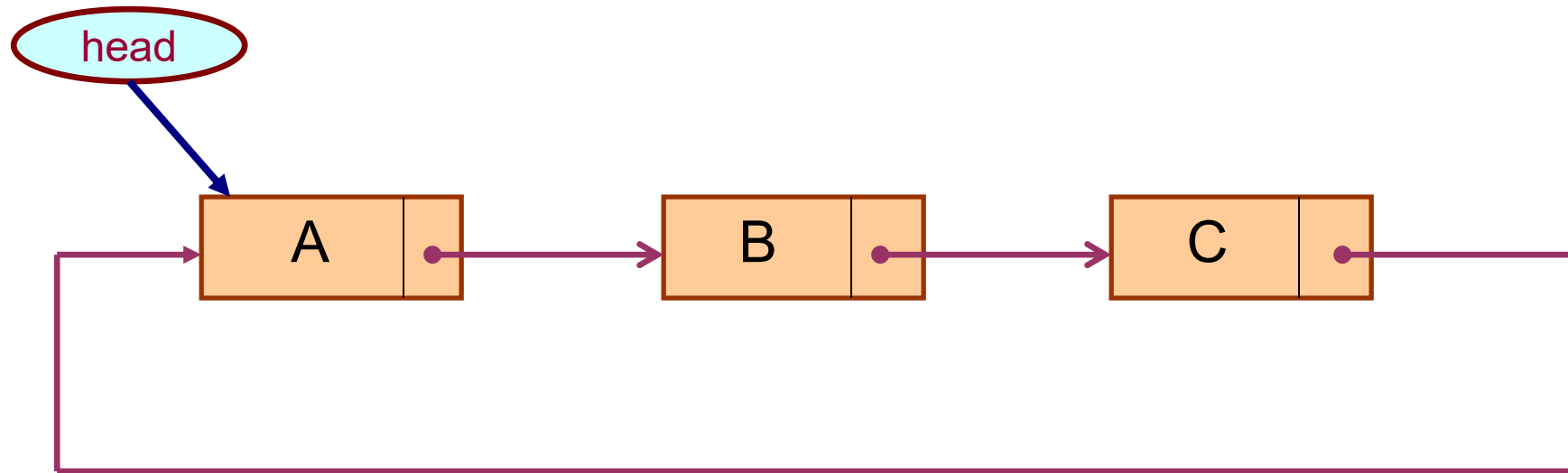
Types of Linked lists

- We will consider these for
  - Singly linked lists
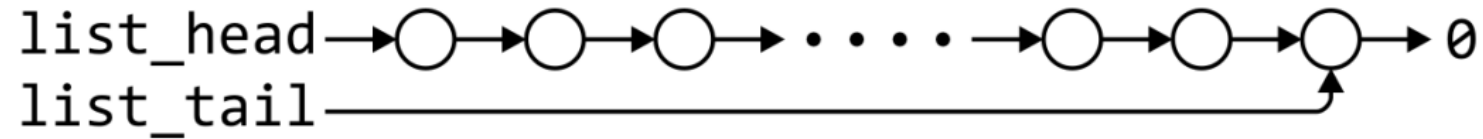
  - Doubly linked lists
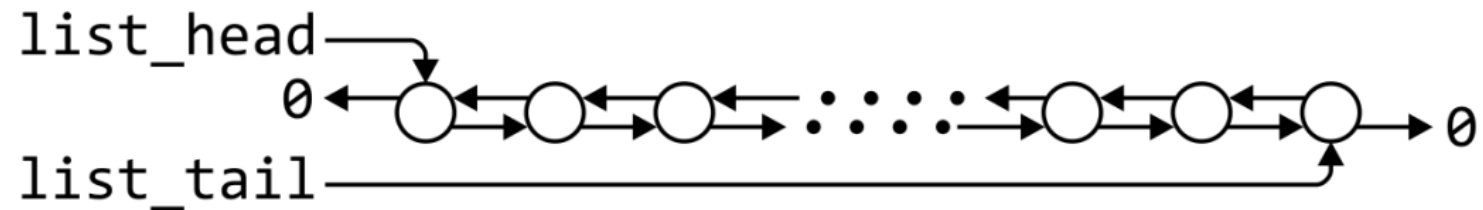
Types of Linked lists

- Doubly Linked List
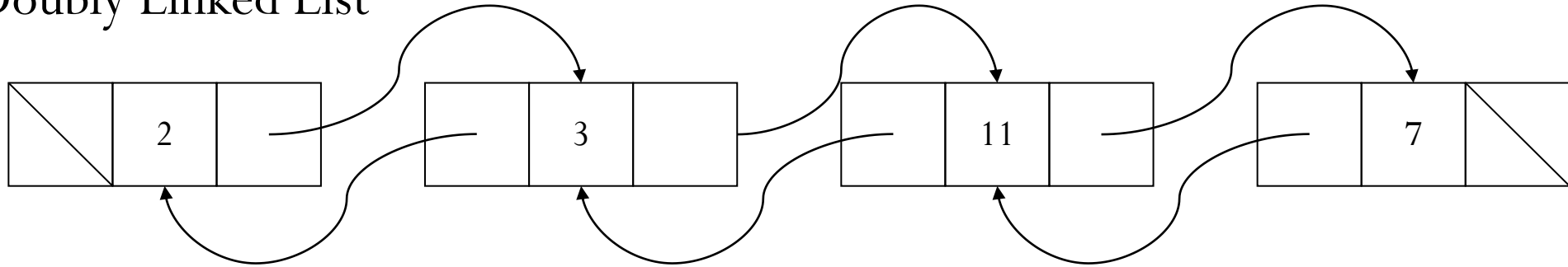
- Circular List
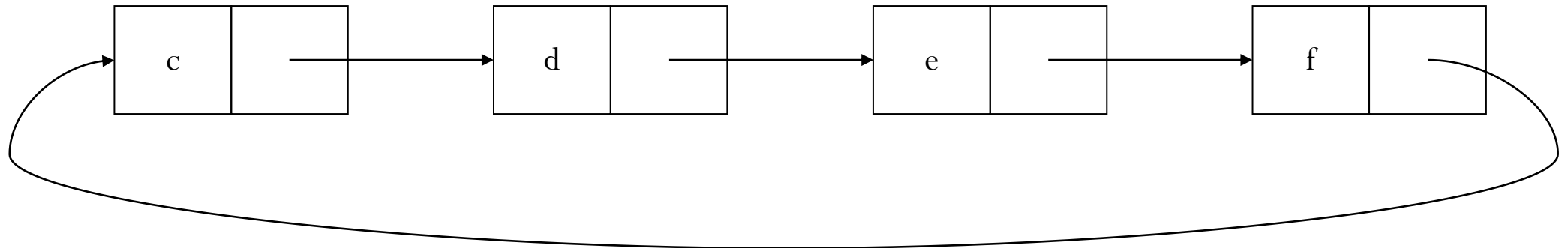
Advantage of Linked List

- An Abstract List (or List ADT) is linearly ordered data where the programmer explicitly defines the ordering

- Most common operations and implementation uses either an Array or Linked list

- Arrays can be used to store linear data of similar types, with following limitations.
  - The size of the arrays is fixed
  - Inserting a new element in an array of elements is expensive

- Advantages over arrays
  - Dynamic size
  - Ease of insertion/deletion

- Drawbacks of linked list
  - Random access is not allowed. We cannot do binary search with linked lists.
  - Extra memory space for a pointer is required with each element of the list.

Dynamic Memory Management

Memory Allocation

The constructor is called whenever an object is created, either:

Static allocation,

List ls;

defines ls to be a linked list and the compiler deals with memory allocation

Dynamic allocation

List *pls = new List();

requests sufficient memory from the OS to store an instance of the class

In both cases, the memory is allocated and then the constructor is called

Pseudo-code for insertion

```
typedef struct nd {
    struct item data;
    struct nd * next;
    } node;


void insert(node *curr)
{
node * tmp;


tmp=(node *) malloc(sizeof(node));
tmp->next=curr->next;
curr->next=tmp;
}
```


Item to be inserted
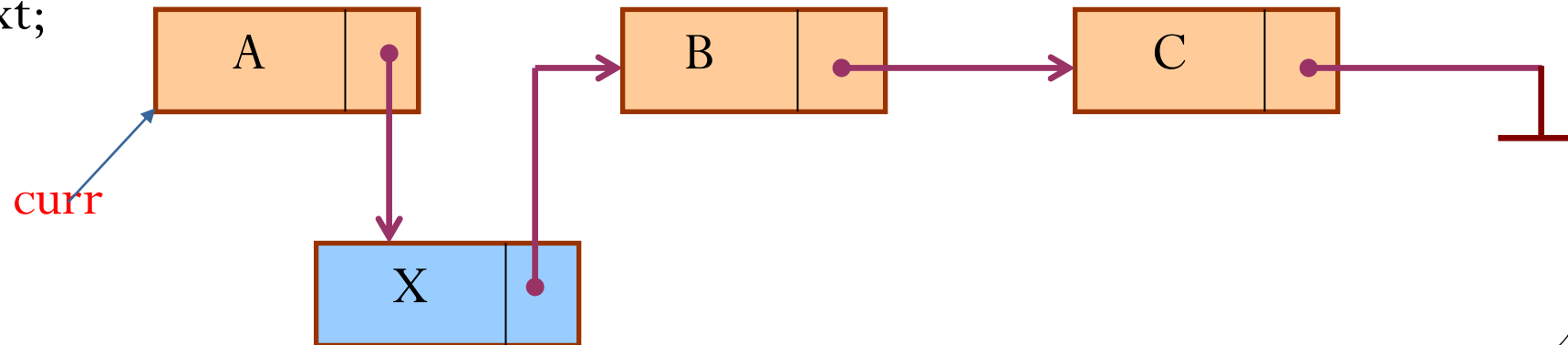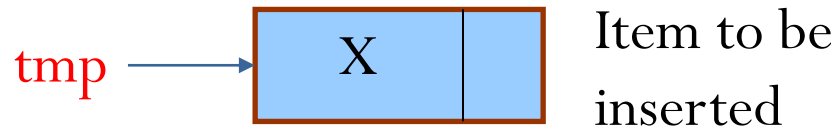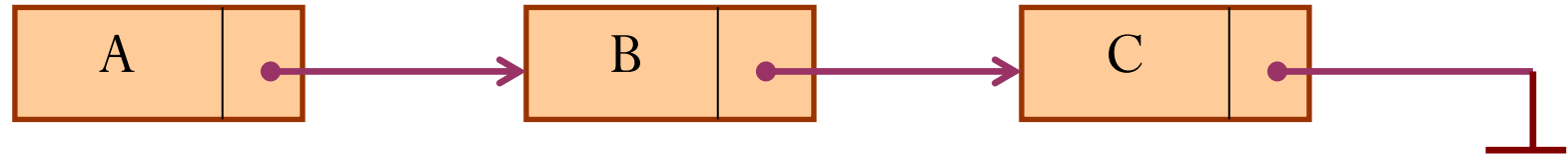
Pseudo-code for deletion

```
typedef struct nd {
    struct item data;
    struct nd * next;
    } node;


void delete(node *curr)
{
node * tmp;
tmp=curr->next;
curr->next=tmp->next;
free(tmp);
}
```

Item to be deleted

curr   tmp
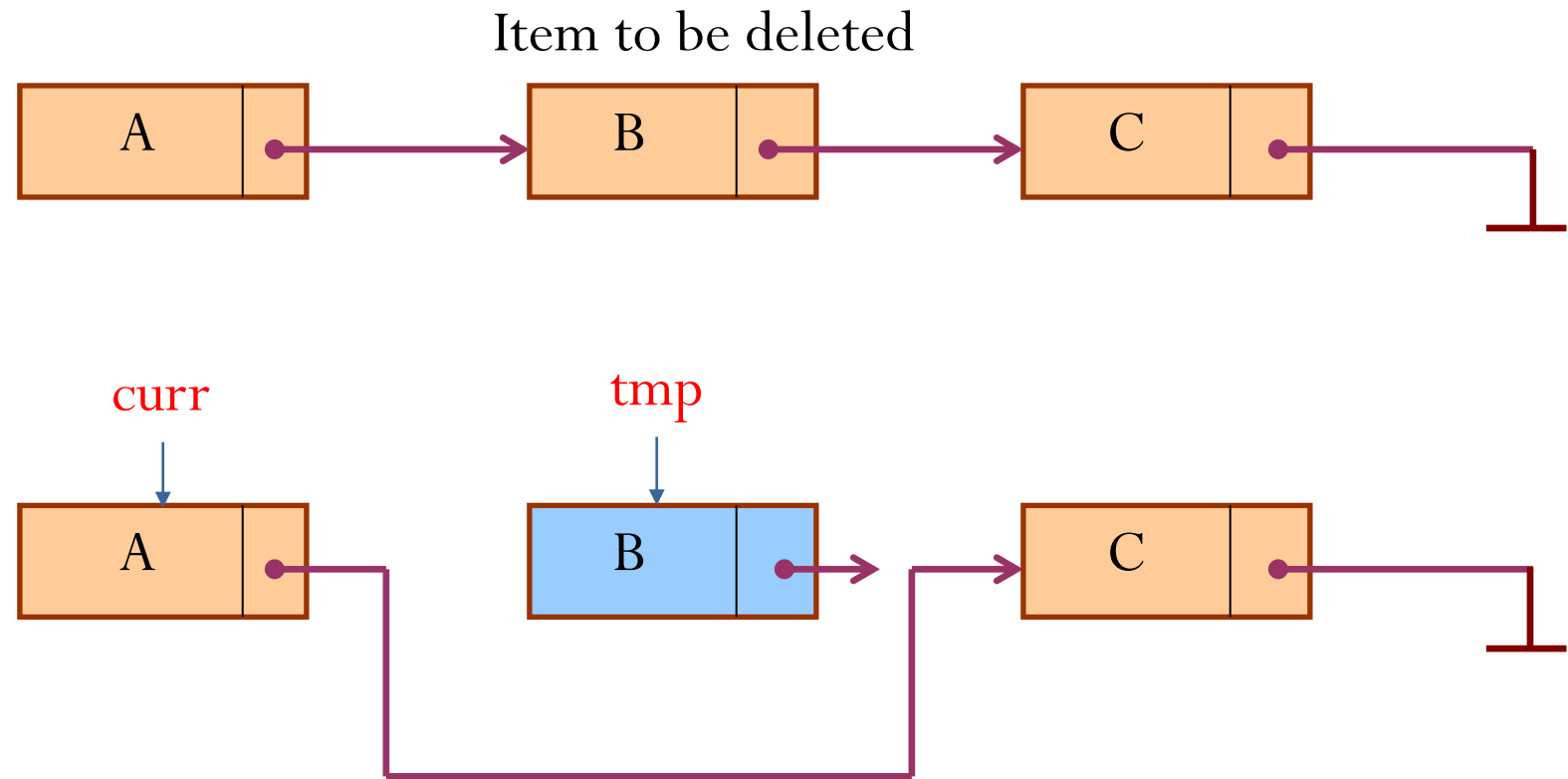
Pseudo-code

- For insertion:
    - A record is created holding the new item.
    - The next pointer of the new record is set to link it to the item which is to follow it in the list.
    - The next pointer of the item which is to precede it must be modified to point to the new item.
- For deletion:
    - The next pointer of the item immediately preceding the one to be deleted is altered, and made to point to the item following the deleted item.

Dynamic Storage: Memory Management

A linked list uses linked allocation, and therefore each node may appear anywhere in memory

Assuming a 32-bit machine

Memory required for each node equals the memory required by the member variables

- 4 bytes for the linked list (a pointer)
- 8 bytes for each node (an **int** and a pointer)

- List could occupy memory
- The **next_node** pointers store the addresses of the next node in the list
- The addresses are arbitrary

0x00572530   42   element
0x00572534   0x010538F0   next_node

0x010538F0   95   element
0x010538F4   0x0105CD28   next_node

Linked List Object

0xFFFF37A0   0x00572530   list_head

0x0073AB40   81   element
0x0073AB44   0x00000000   next_node

0x0105CD28   70   element
0x0105CD2C   0x0073AB40   next_node

0
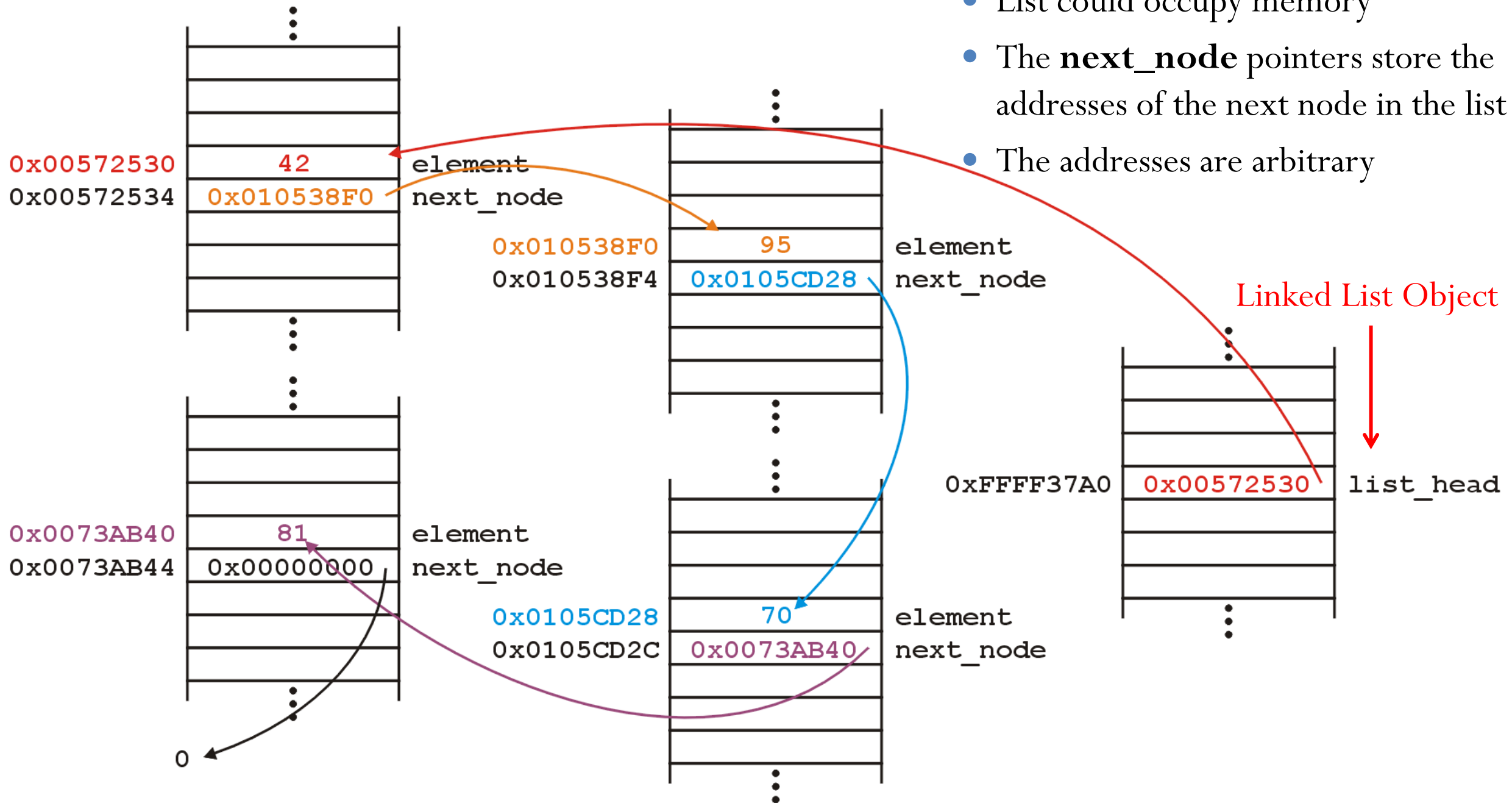
Dynamic Storage: Memory Management

- Clean representation as follows:

list_head ——————→ (42) ——————→ (95) ——————→ (70) ——————→ (81) ——————→ 0

- We do not specify the addresses because they are arbitrary and:
  - The contents of the circle is the value
  - The next_node pointer is represented by an arrow

Applications of Linked List

Ordered List Examples

- MONDAY, TUEDSAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY

- 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King, Ace

- 1941, 1942, 1943, 1944, 1945

- a_1, a_2, a_3, …, a_{n-1}, a_n

- ordered (linear) list: (item1, item2, item3, …, item$n$)

Applications

- Everything!
  - Class list
  - compilers: list of functions in a program, statements in a function
  - graphics: list of triangles to be drawn to the screen
  - operating systems: list of programs running
  - music: compose crazy hard transcendental études
  - other data structures: queues, stacks!

Application: Polynomial ADT

A_i is the coefficient of the x^{n-i} term:

$3x^2 + 2x + 5$ **(3 2 5)**

$8x + 7$ **(8 7)**

$x^2 + 3$ **(1 0 3)**

Problem?

$$x^{2001} + 1$$

(1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1)

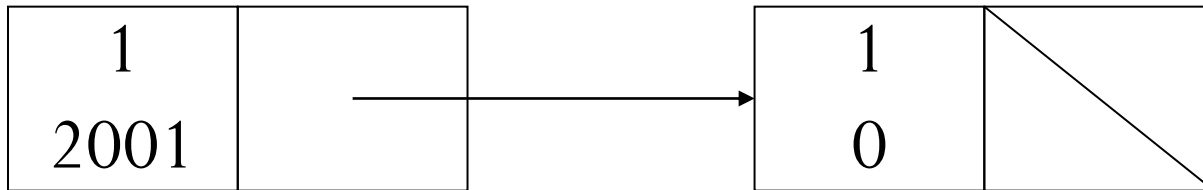# Sparse List Data Structure (?): $x^{2001} + 1$

(<1 2001> <1 0>)

Linked List   vs.   Array

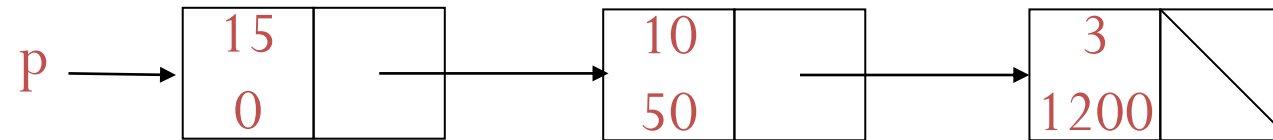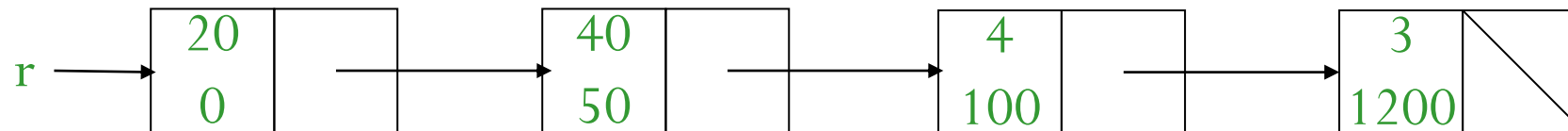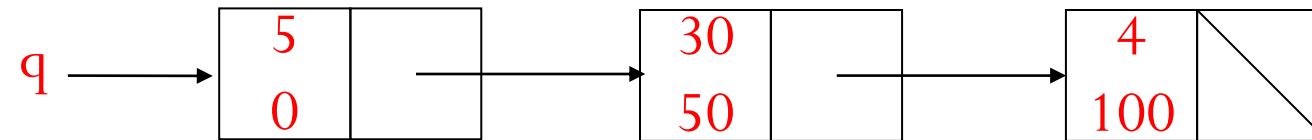| 1		1		1	1
2001		0		2001	0

Addition of Two Polynomials

- Similar to merging two sorted lists

$15+10x^{50}+3x^{1200}$

$5+30x^{50}+4x^{100}$

Multiple Linked Lists

- Many ADTS such as graphs, relations, sparse matrices, multivariate polynomials use multiple linked lists

- Several options
  - array of lists
  - lists of lists
  - multi lists

- General principle: use one ADT to implement a more complicated one.

Array of Linked Lists: Adjacency List for Graphs

- Array G of unordered linked lists

- Each list entry corresponds to an edge in the graph
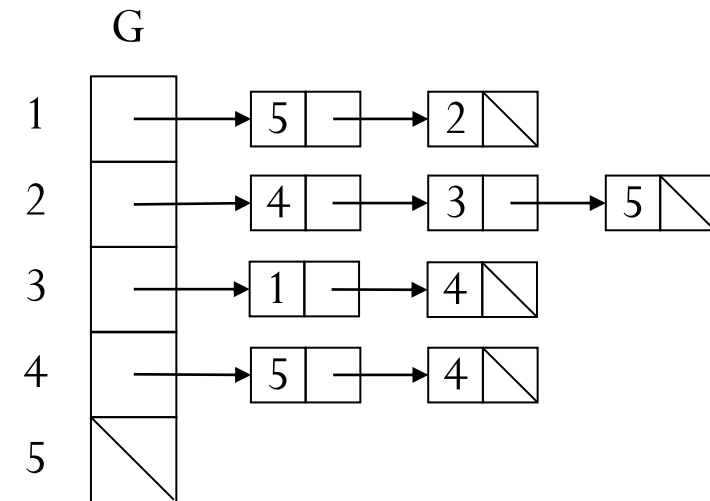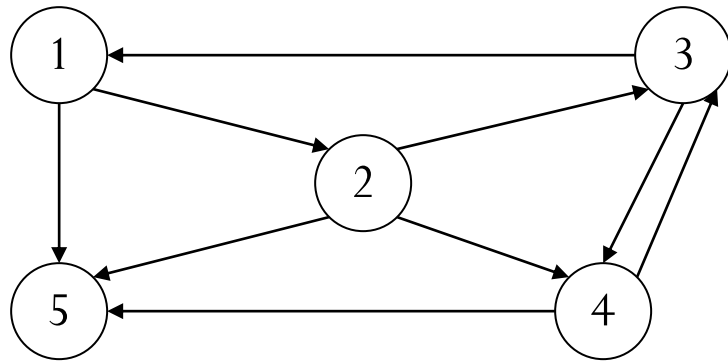
Reachability by Marking

- Suppose we want to mark all the nodes in the graph which are reachable from a given node **k**.
  - Let **G[1..n]** be the adjacency list rep. of the graph
  - Let **M[1..n]** be the mark array, initially all **false**s.

```
mark(int i){
  M[i] = true;
  x = G[i]
  while (x != NULL) {
    if (M[x->node] == false)
      mark(G[x->node])
    x = x->next
  }
}
```
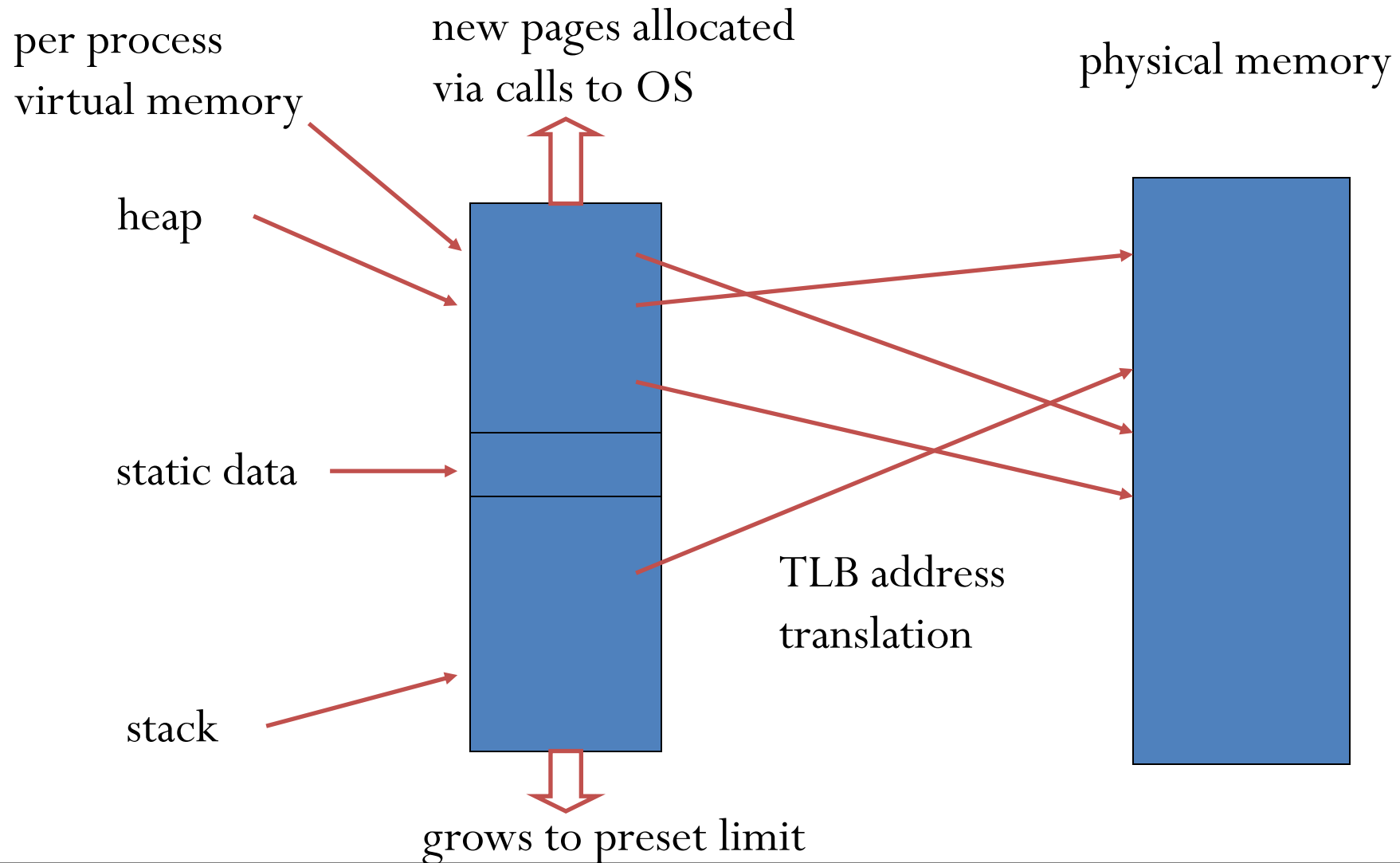
Thoughts on Reachability

- The marking algorithm visits each node and each edge at most once.

- This marking algorithm uses Depth First Search. DFS uses a stack to track nodes. Where?

- Graph reachability is closely related to garbage collection
  - the nodes are blocks of memory
  - marking starts at all global and active local variables
  - the marked blocks are reachable from a variable
  - unmarked blocks are garbage

Garbage Collection

Garbage Collection

- Every modern programming language allows programmers to allocate new storage dynamically
  - New records, arrays, tuples, objects, closures, etc.

- Every modern language needs facilities for reclaiming and recycling the storage used by programs

- It's usually the most complex aspect of the run-time system for any modern language (Java, ML, Lisp, Scheme, Modula, …)

Memory layout

per process
virtual memory

new pages allocated
via calls to OS

physical memory

heap

static data

stack

TLB address
translation

grows to preset limit

Garbage Collection

- What is garbage?
  - A value is garbage if it will not be used in any subsequent computation by the program

- Is it easy to determine which objects are garbage?
  - No. It's undecidable. Eg:

    if long-and-tricky-computation then use v

    else don't use v

Garbage Collection

- Since determining which objects are garbage is tricky, people have come up with many different techniques
  - It's the programmers problem:
    - Explicit allocation/deallocation
  - Reference counting
  - Tracing garbage collection
    - Mark-sweep, copying collection
    - Generational Garbage Collection

Explicit Memory Management

- User library manages memory; programmer decides when and where to allocate and deallocate
  - void* malloc(long n)
  - void free(void *addr)
  - Library calls OS for more pages when necessary
  - Advantage: people are smart
  - Disadvantage: people are dumb and they really don't want to bother with such details if they can avoid it

Explicit Memory Management

- How does malloc/free work?
  - Blocks of unused memory stored on a freelist
  - malloc: search free list for usable memory block
  - free: put block onto the head of the freelist
- Drawbacks
  - malloc is not free: we might have to do a significant search to find a big enough block
 - As program runs, the heap fragments leaving many small, unusable pieces

Automatic Memory Management (MM)

- Languages with Explicit MM are much harder to program than languages with Automatic MM
  - Always worrying about dangling pointers, memory leaks: a huge software engineering burden
  - Impossible to develop a secure system, impossible to use these languages in emerging applications involving mobile code
  - Soon, languages with unsafe, Explicit MM will all but disappear

Automatic Memory Management

- Question: how do we decide which objects are garbage?
  - We conservatively approximate
  - Normal solution: an object is garbage when it becomes unreachable from the roots
    - The roots = registers, stack, global static data
    - If there is no path from the roots to an object, it cannot be used later in the computation so we can safely recycle its memory

Object Graph

stack

r1

r2
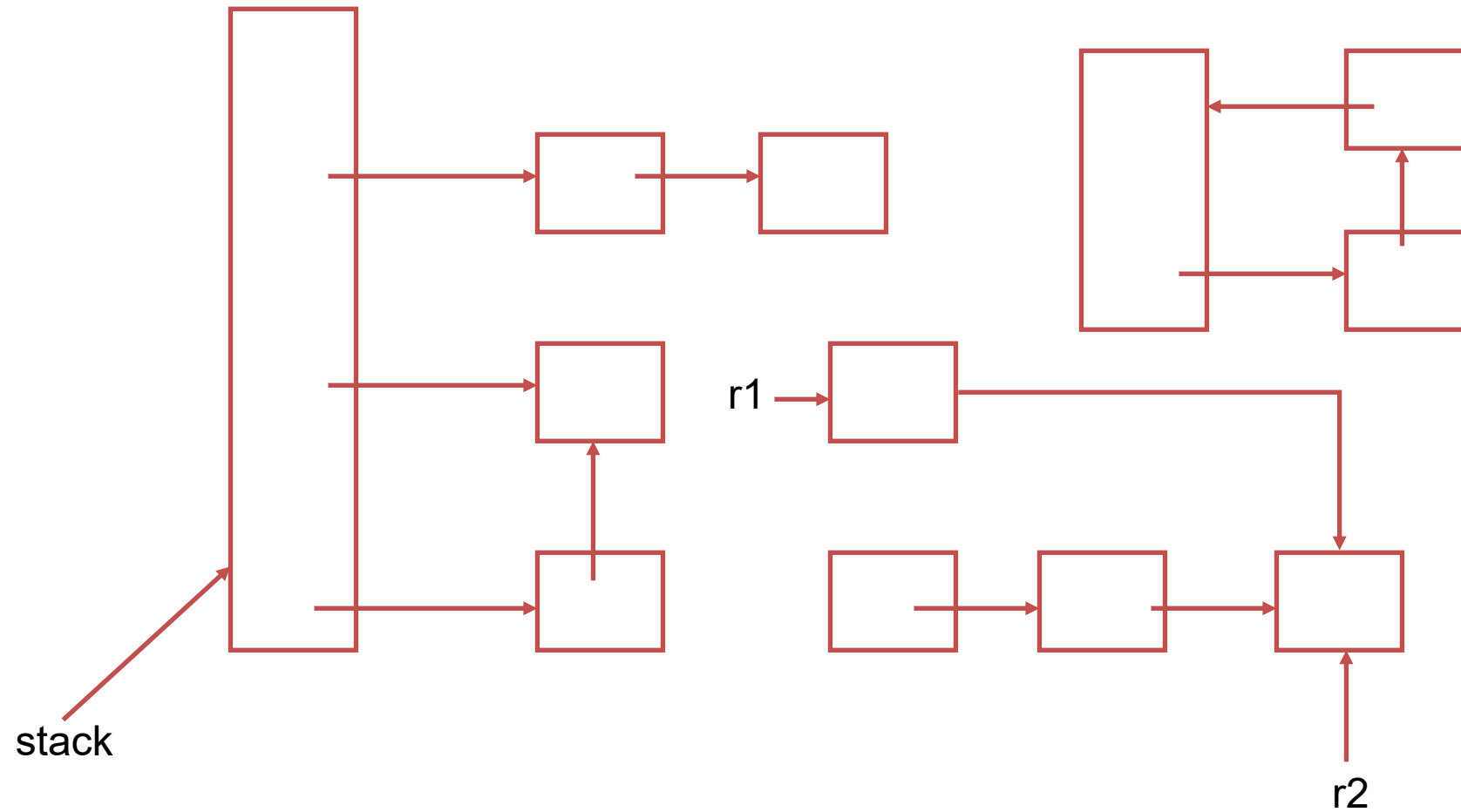
- How should we test reachability?

Object Graph

Reference counting can't detect cycles

stack

r1

r2
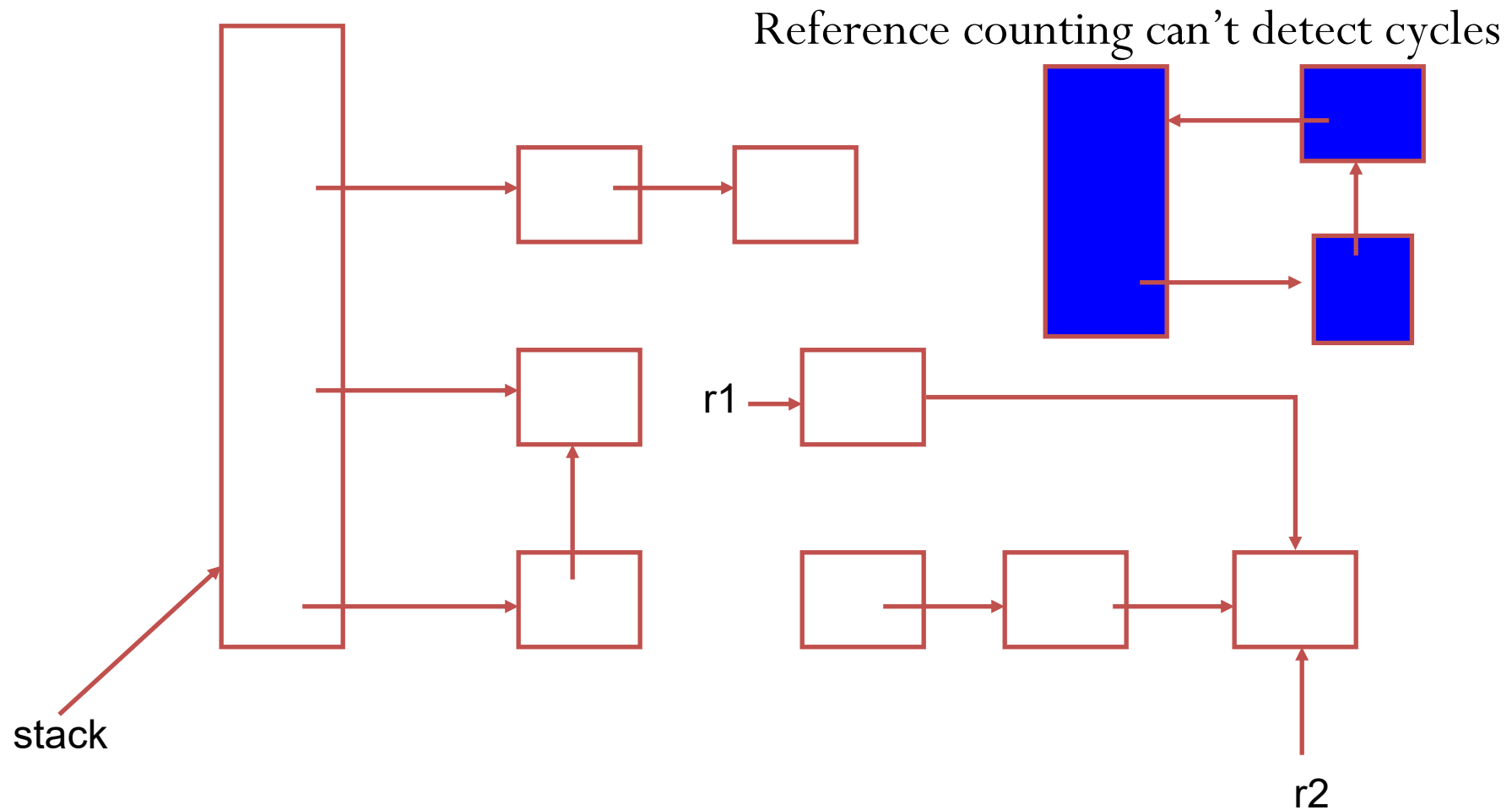
- Keep track of the number of pointers to each object (the reference count).
- When the reference count goes to 0, the object is unreachable garbage

Copying Collection

- Basic idea: use 2 heaps
  - One used by program
  - The other unused until GC time
- Garbage Collection:
  - Start at the roots & traverse the reachable data
  - Copy reachable data from the active heap (from-space) to the other heap (to-space)
  - Dead objects are left behind in from space
  - Heaps switch roles

Student Linked List

Student Linked List

- Consider the structure of a node as follows:

```
struct stud {
            int    roll;
            char   name[25];
            int    age;
            struct stud *next;
        };


  /* A user-defined data type called "node" */

typedef struct stud node;
node *head;
```

Student Linked List

- To start with, we have to create a node (the first node), and make `head` point to it

  `head = (node *) malloc(sizeof(node));`

- If there are n number of nodes in the initial linked list:

  - Allocate n records, one by one.

  - Read in the fields of the records.

  - Modify the links of the records so that the chain is formed.
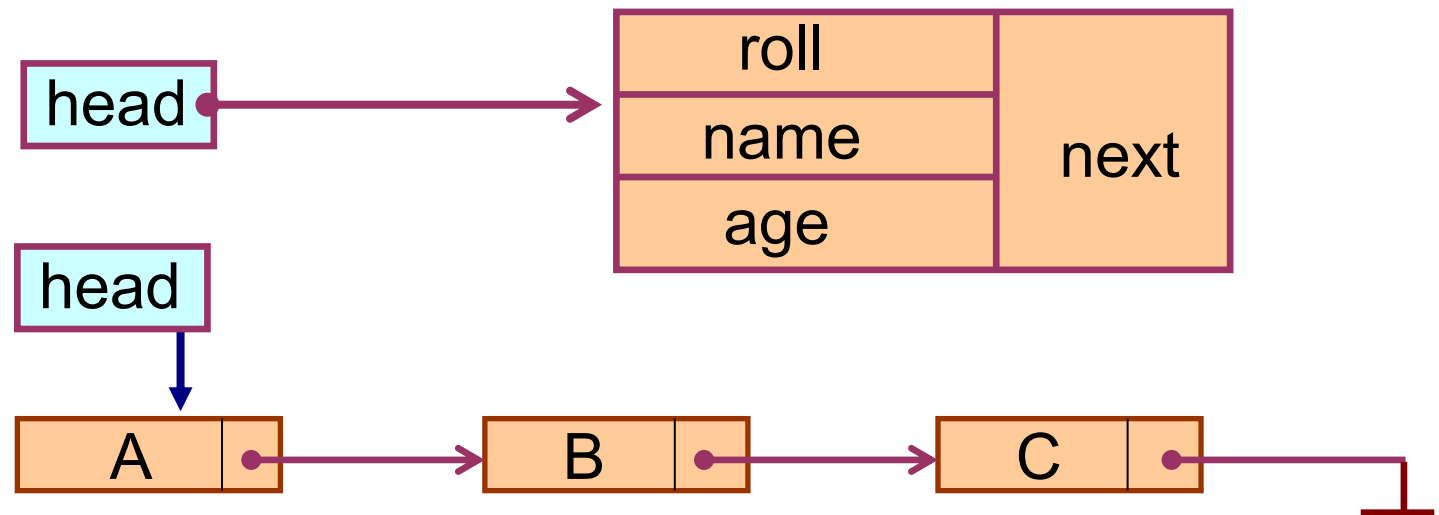
Create the Student Linked List

To be called from `main()` function as:

```
node *head;
.........
head = create_list();
```

```c
node *create_list()
{
    int  k, n;
    node  *p, *head;

    printf  ("\n How many elements to enter?");
    scanf ("%d", &n);

    for  (k=0; k<n; k++)
    {   if (k == 0) {
            head = (node *) malloc(sizeof(node));
            p = head;
        }
        else {
            p->next  = (node *) malloc(sizeof(node));
            p = p->next;
        }
        scanf ("%d %s %d", &p->roll, p->name, &p->age);
    }

    p->next  =  NULL;
    return (head);
}
```

Traverse the Student Linked List

```
void display (node *head)
{
  int  count = 1;
  node  *p;

  p = head;
  while (p != NULL)
  { printf ("\nNode %d: %d %s %d", count, p->roll, p->name, p->age);
    count++;
    p = p->next;
  }
  printf ("\n");
}
```

- To be called from `main()` function

```
node *head;
    ………
display (head);
```

- After construction, *head* points to the first node of the list,
  - Follow the pointers.
  - Display the contents of the nodes as they are traversed.
  - Stop when the *next* pointer points to NULL.

Inserting a Node in a List

Inserting a Node in the Student Linked List

- The problem is to insert a node *before a specified node*.
  - Specified means some value is given for the node (called *key*).
  - In this example, we consider it to be `roll`.

- Convention followed:
  - If the value of roll is given as *negative*, the node will be inserted at the *end* of the list.

- To be called from `main()` function

```
void insert (node **head)
{
    int  k = 0, rno;
    node *p, *q, *new;

    new = (node *) malloc(sizeof(node));

    printf ("\nData to be inserted: ");
      scanf ("%d %s %d", &new->roll, new->name, &new->age);
    printf ("\nInsert before roll (-ve for end):");
      scanf ("%d", &rno);

    p = *head;
```

```
node *head;

………

insert (&head);
```

Insert a Node in the Student Linked List

- When a node is added at the beginning,
  - Only one next pointer needs to be modified.
    - *head* is made to point to the new node.
    - New node points to the previously first element.
- When a node is added at the end,
  - Two next pointers need to be modified.
    - Last node now points to the new node.
    - New node points to NULL.
- When a node is added in the middle,
  - Two next pointers need to be modified.
    - Previous node now points to the new node.
    - New node points to the next node.
- The pointers q and p always point to consecutive nodes.

```
if (p->roll == rno)    /* At the beginning */
   {   new->next = p;
       *head = new;
   }
else
   {
   while ((p != NULL) && (p->roll != rno))
       {   q = p;
           p = p->next;
       }

       if  (p == NULL)    /* At the end */
       {   q->next = new;
           new->next = NULL;
       }
   else if  (p->roll == rno)
               /* In the middle */
           {   q->next = new;
               new->next = p;
           }
   }
}
```

Delete a Node in the Student Linked List

```c
void  delete (node **head)
{    int   rno;
     node   *p, *q;

     printf ("\nDelete for roll :");
      scanf ("%d", &rno);


     p = *head;
```

- Required to delete a specified node.
  - Roll Number is given.
- Here also three conditions arise:
  - Deleting the first node.
  - Deleting the last node.
  - Deleting an intermediate node.

```c
if  (p->roll == rno) /* Delete the first element */
     {    *head = p->next;
          free (p);
     }


else
     {  while  ((p != NULL) && (p->roll != rno))
         {    q = p;
              p  =  p->next;
         }

          if  (p == NULL)    /* Element not found */
            printf ("\nNo match deletion failed");

          else if (p->roll == rno)
                  /* Delete any other element */
             {    q->next  =  p->next;
                  free (p);
             }
     }
}
```

Exercises

- Write a function to:
  - Concatenate two given list into one big list.

    node *concatenate (node *head1, node *head2);

  - Insert an element in a linked list in sorted order. The function will be called for every element to be inserted.

    void insert_sorted (node **head, node *element);

  - Always insert elements at one end, and delete elements from the other end (first-in first-out QUEUE).

    void insert_q (node **head, node *element)

    node *delete_q (node **head)   /* Return the deleted node */

References

- Donald E. Knuth, The Art of Computer Programming, Volume 1: Fundamental Algorithms, 3rd Ed., Addison Wesley, 1997.
- Cormen, Leiserson, and Rivest, Introduction to Algorithms, McGraw Hill, 1990.
- Weiss, Data Structures and Algorithm Analysis in C++, 3rd Ed., Addison Wesley.
- Koffman and Wolfgang, "Objects, Abstraction, Data Structures and Design using C++", John Wiley & Sons, Inc., Ch. 6.
- David Walker CS 320, "Garbage collection" https://www.cs.princeton.edu/courses/archive/spr03/cs320/notes/gc.ppt
- Wikipedia, http://en.wikipedia.org/wiki/Double-ended_queue
- CSE326: Data Structure, Department of Computer Science and Engineering, University of Washington https://courses.cs.washington.edu/courses/cse326
- Mike Scott, CS 307 Fundamentals of Computer Science, https://www.cs.utexas.edu/~scottm/cs307/
- Debasis Samanta, Computer Science & Engineering, Indian Institute of Technology Kharagpur, Spring-2017, Programming and Data Structures. https://cse.iitkgp.ac.in/~dsamanta/courses/pds/index.html

ขอบคุณ
Thai

Grazie
Italian

תודה רבה
Hebrew

ಧನ್ಯವಾದಗಳು
Kannada

धन्यवादः
Sanskrit

Ευχαριστώ
Greek

Thank You
English

Gracias
Spanish

Спасибо
Russian

Obrigado
Portuguese

شكراً
Arabic

https://sites.google.com/site/animeshchaturvedi07

Merci
French

多謝
Traditional
Chinese

धन्यवाद
Hindi

Danke
German

多谢
Simplified
Chinese

நன்றி
Tamil

Tamil

ありがとうございました
Japanese

감사합니다
Korean