CS - 114 : Computer Workshop

Prof. Chamakuri Nagaiah
Mahindra-École Centrale, Hyderabad
nagaiah.chamakuri@mechyd.ac.in

Linked List :: Basic Concepts

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

  – An array is an example of a list. The array index is used for accessing and manipulation of array elements.

- Problems with array:

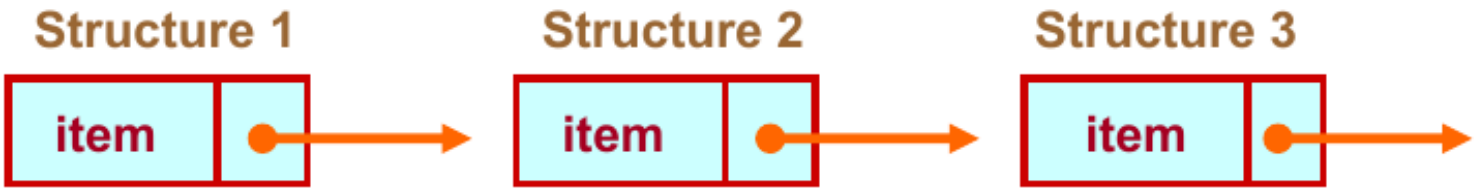
Linked List :: Basic Concepts

- A linked list is a sequence of data structures, which are connected together via links.
  – An array is an example of a list. The array index is used for accessing and manipulation of array elements.
- Problems with array:
  – Fast element access (+)
  – The array size has to be specified at the beginning and impossible to resize(-).
  – Deleting an element or inserting an element may require shifting of elements (-). Required size not always immediately available.
- A completely different way to represent a list:

Linked List :: Basic Concepts

- A linked list is a sequence of data structures, which are connected together via links.
  – An array is an example of a list. The array index is used for accessing and manipulation of array elements.
- Problems with array:
  – Fast element access (+)
  – The array size has to be specified at the beginning and impossible to resize(-).
  – Deleting an element or inserting an element may require shifting of elements (-). Required size not always immediately available.
- A completely different way to represent a list:
  – Make each item in the list part of a structure.
  – The structure also contains a pointer or link to the structure containing the next item.
  – This type of list is called a linked list.

Linked List :: Basic Concepts

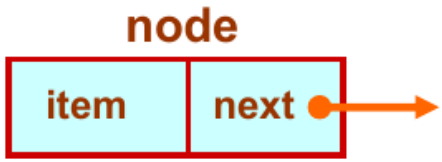


- Each structure of the list is called a node, and consists of two fields:
  - One containing the item.
  - The other containing the address of the next item in the list.
- The data items comprising a linked list need not be contiguous in memory.
  - They are ordered by logical links that are stored as part of the data in the structure itself.
  - The link is a pointer to another structure of the same type.

Linked List :: Basic Concepts

- Such a structure can be represented as:

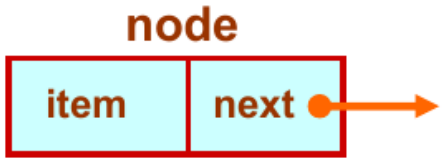
```
struct node {
  int item;
  struct node *next;
};
```



Linked List :: Basic Concepts

- Such a structure can be represented as:

```
struct node {
  int item;
  struct node *next;
};
```

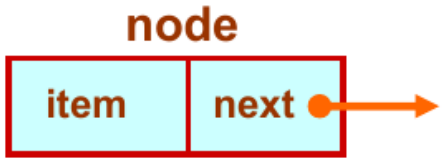


- Such structures which contain a member field pointing to the same structure type are called self-referential structures.

Linked List :: Basic Concepts

- Such a structure can be represented as:

```
struct node {
  int item;
  struct node *next;
};
```



- Such structures which contain a member field pointing to the same structure type are called self-referential structures.
- In general, a node may be represented as follows:

```
struct node_name{
  type member1; type member2;
  .........
  struct node_name *next;
};
```

Example

- Consider the structure:

```
struct stud
  {
    int roll;
    char name[30];
    int age;
    struct stud *next;
  };
```

Example

- Consider the structure:

```
struct stud
  {
    int roll;
    char name[30];
    int age;
    struct stud *next;
  };
```

- Also assume that the list consists of three nodes n1, n2 and n3.

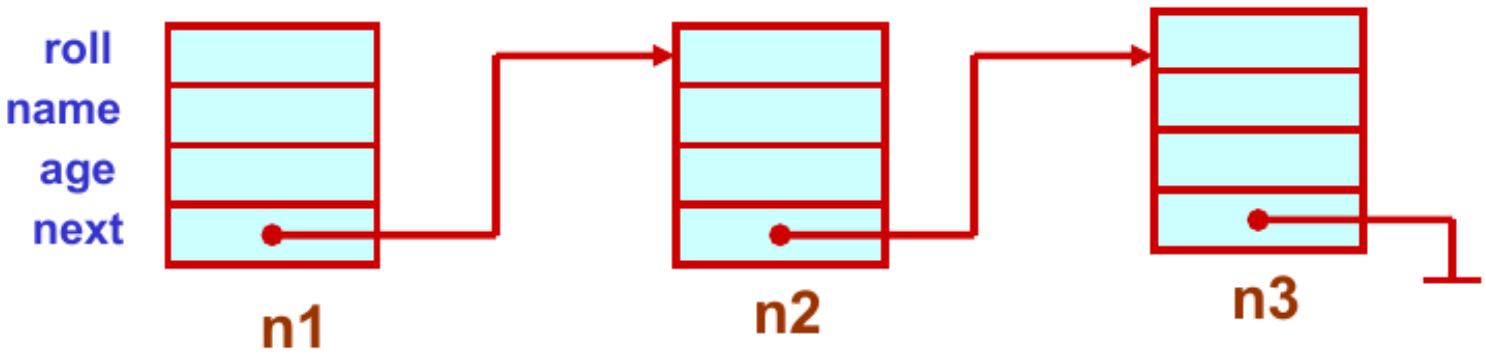
  struct stud n1, n2, n3;

Example

- To create the links between nodes, we can write:

```
n1.next = &n2 ;
n2.next = &n3 ;
n3.next = NULL ; /* No more nodes follow */
```

- Now the list looks like:



Example

```
#include <stdio.h>
struct stud {
  int roll;
  char name[30];
  int age;
  struct stud *next;
 };
main() {
  struct stud n1, n2, n3;
  struct stud *p;
  scanf ("%d %s %d", &n1.roll, n1.name, &n1.age);
  scanf ("%d %s %d", &n2.roll, n2.name, &n2.age);
  scanf ("%d %s %d", &n3.roll, n3.name, &n3.age);
```

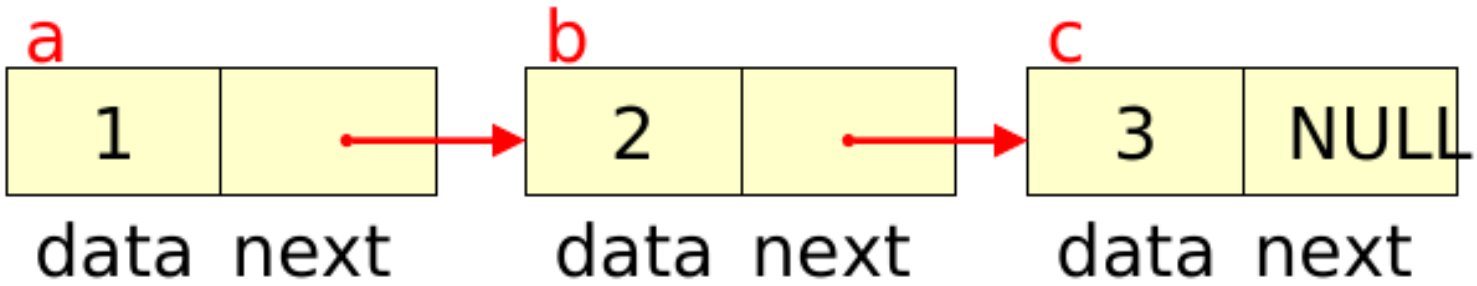
Example

```
  n1.next = &n2 ;
  n2.next = &n3 ;
  n3.next = NULL ;
  /* Now traverse the list and print the elements */

  p = n1 ;
  /* point to 1st element */
  while (p != NULL)
  {
    printf ("\n␣%d␣%s␣%d", p->roll, p->name, p->age);
    p = p->next;
  }
}
```

Accessing data

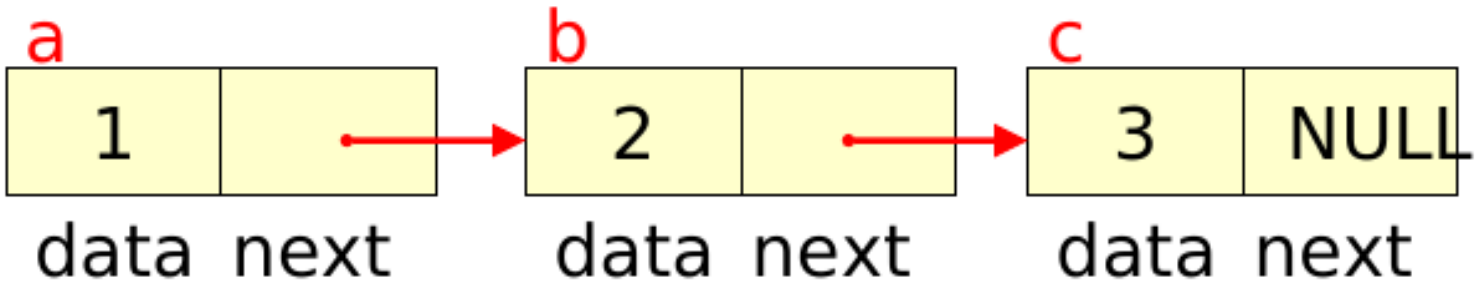
- a.next = &b;
  b.next = &c;



- What are the values of :
  - a.next->data is

Accessing data

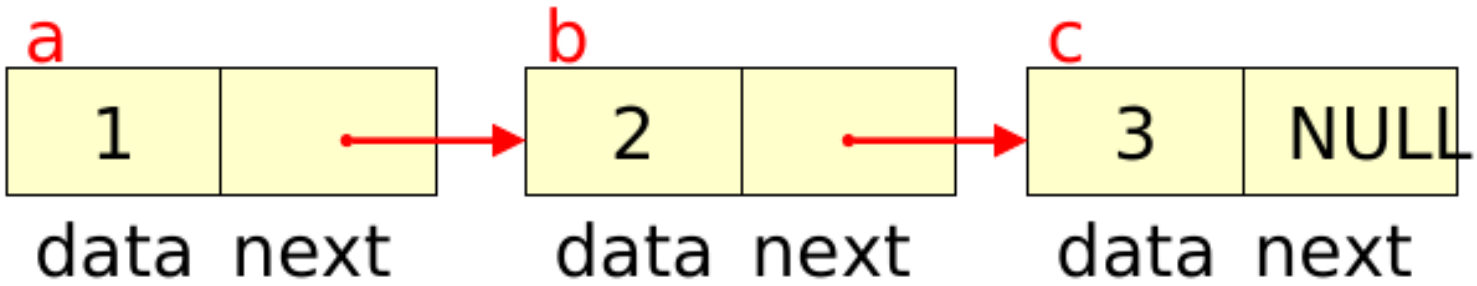
- a.next = &b;
  b.next = &c;



- What are the values of :
  - a.next->data is 2
  - a.next->next->data is

Accessing data

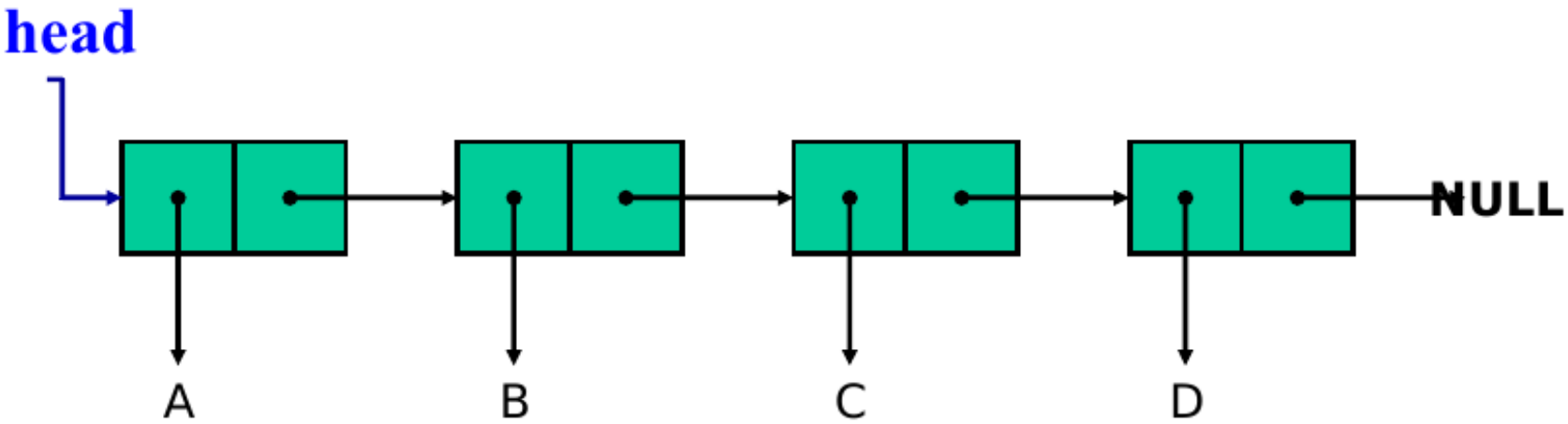
- a.next = &b;
  b.next = &c;



- What are the values of :
  - a.next->data is 2
  - a.next->next->data is 3

Linear Linked Lists

- A head pointer addresses the first element of the list.
- Each element points at a successor element.
- The last element has a link value NULL.



Storage allocation: Say list.h

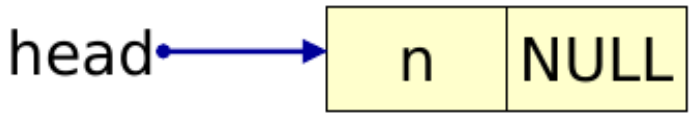
```
#include <stdio.h>
#include <stdlib.h>
typedef char DATA;

struct list {
  DATA d;
  struct list * next;
 };

typedef struct list ELEMENT;
typedef ELEMENT * LINK;
```

Storage allocation

- create a single element list.




```
LINK head ;
head = (LINK) malloc (sizeof(ELEMENT));
head->d = 'n';
head->next = NULL;
```

Storage allocation

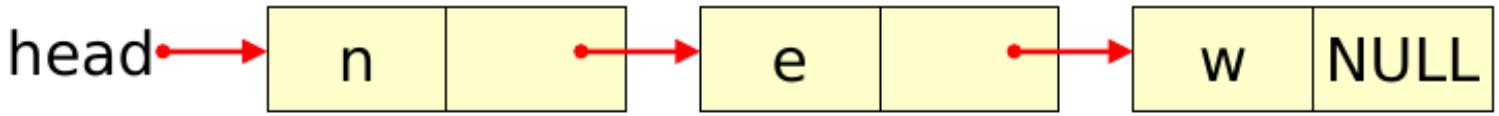
- A second element is added.

- Code:

```
head->next = malloc (sizeof(ELEMENT));
head->next->d = 'e';
head->next->next = NULL;
```

Storage allocation

- We have a 3 element list pointed to by head.
- The list ends when next has the sentinel value NULL.



- Code:

```
head->next->next = malloc (sizeof(ELEMENT));
head->next->next->d = 'w';
head->next->next->next = NULL;
```

List operations

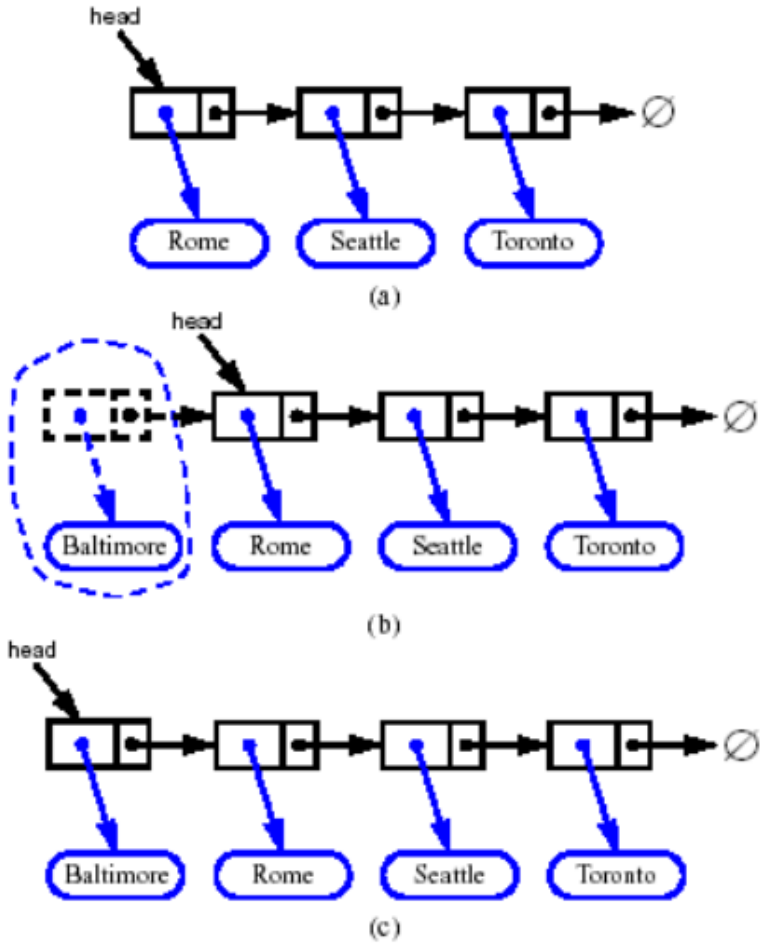
- How to initialize such a self referential structure (LIST),
- how to insert such a structure into the LIST,
- how to delete elements from it,
- how to search for an element in it,
- how to print it,
- how to free the space occupied by the LIST?

list from a string

```
#include "list.h"
LINK StringToList (char s[]) {
LINK head = NULL, tail;
int i;
if (s[0] != '\0') {
  head = malloc (sizeof(ELEMENT));
  head->d = s[0];
  tail = head;
  for (i=1; s[i] != '\0'; i++) {
    tail->next = malloc(sizeof(ELEMENT));
    tail = tail->next;
    tail->d = s[i];
  }
  tail->next = NULL;
 }
return head;
}
```

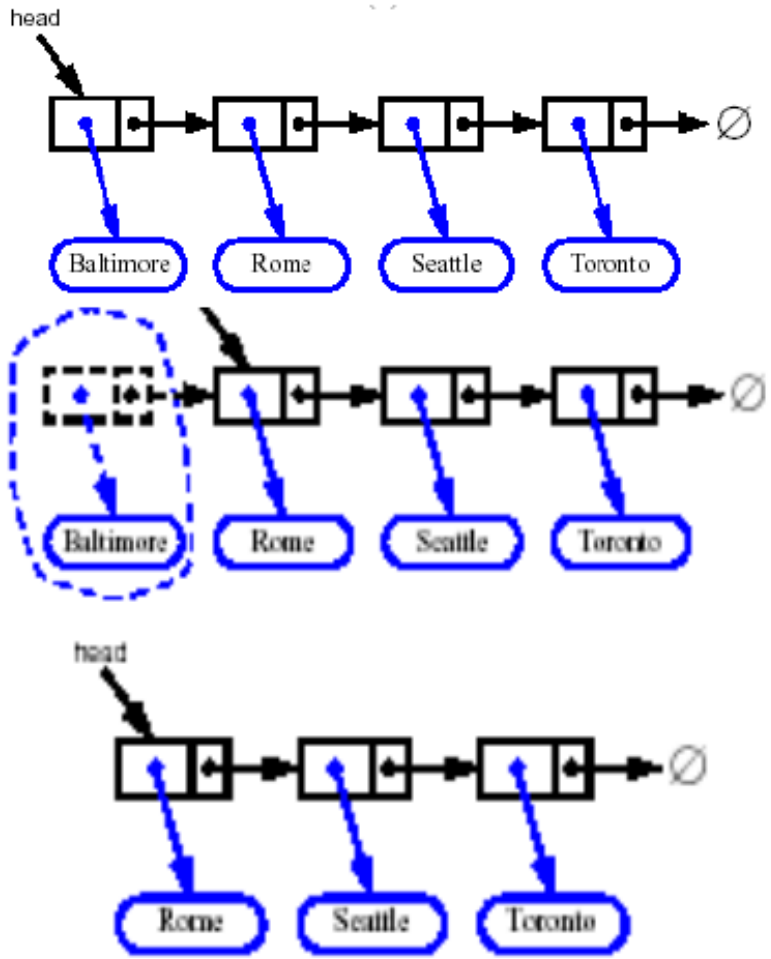
Inserting at the Head

- Allocate a new node
- Insert new element
- Make new node point to old head
- Update head to point to new node



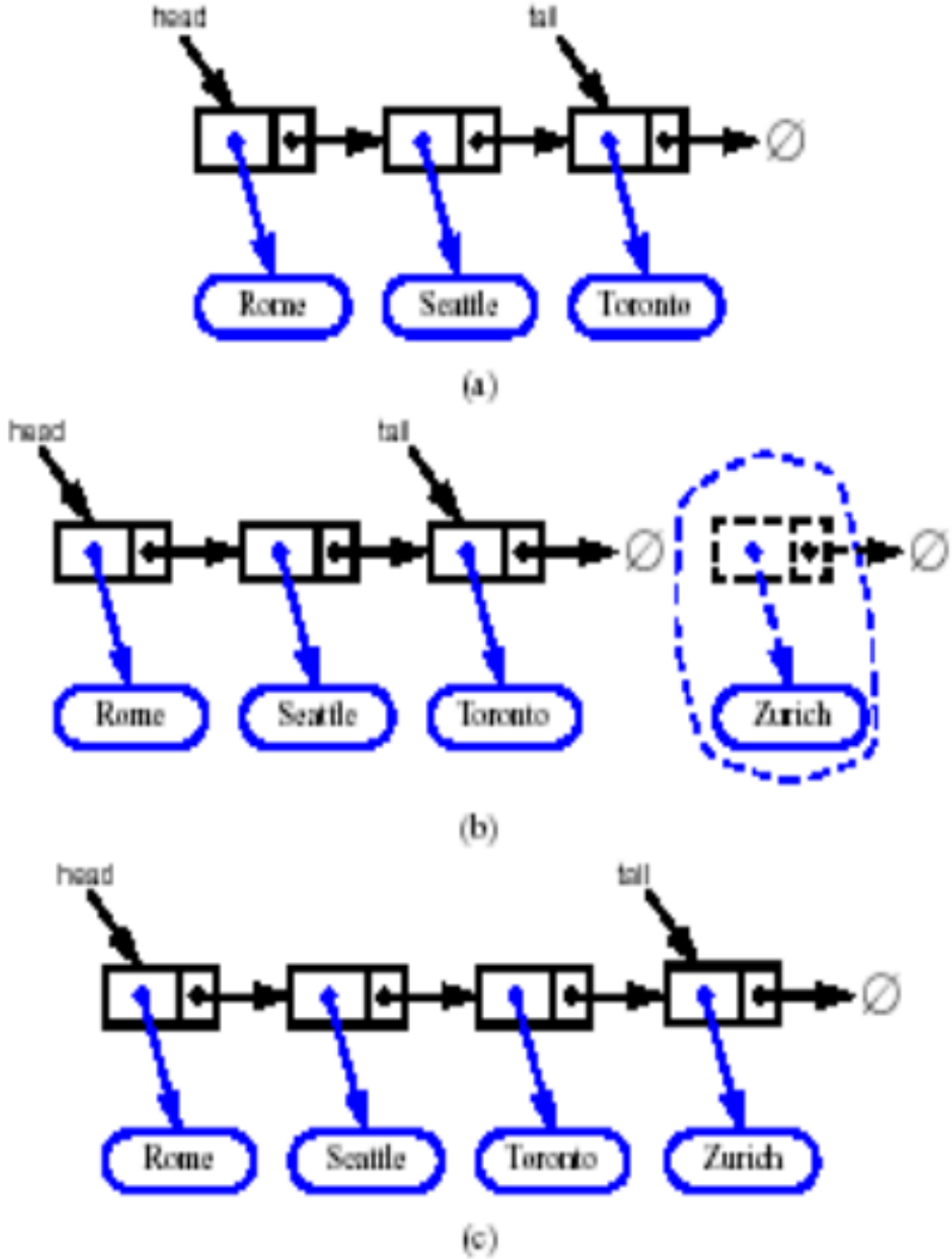
Removing at the Head

- Update head to point to next node in the list
- Allow garbage collector to reclaim the former first node



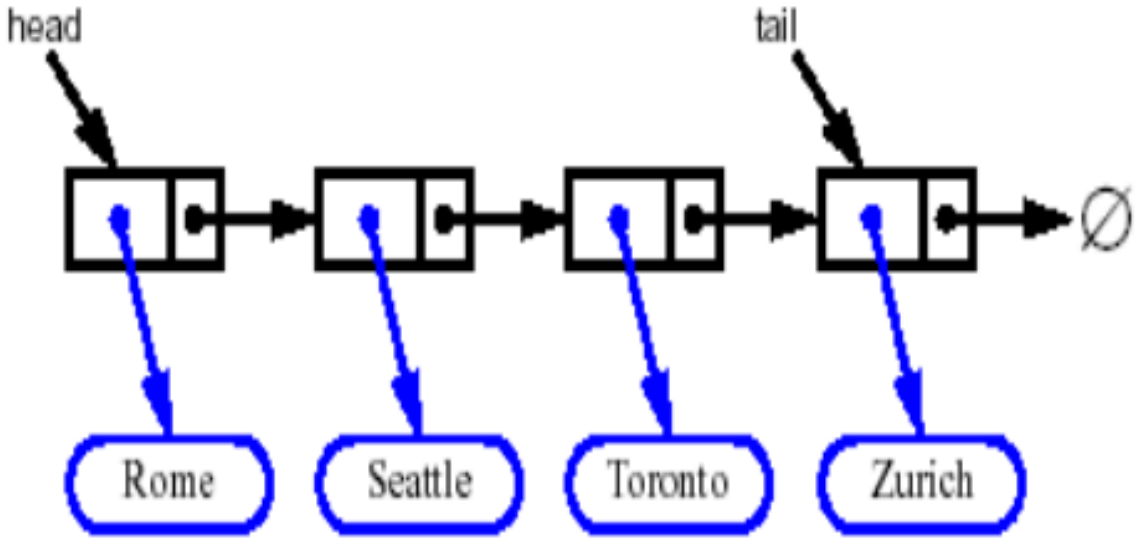
Inserting at the Tail

- Allocate a new node
- Insert new element
- Have new node point to null
- Have old last node point to new node
- Update tail to point to new node



Removing at the Tail

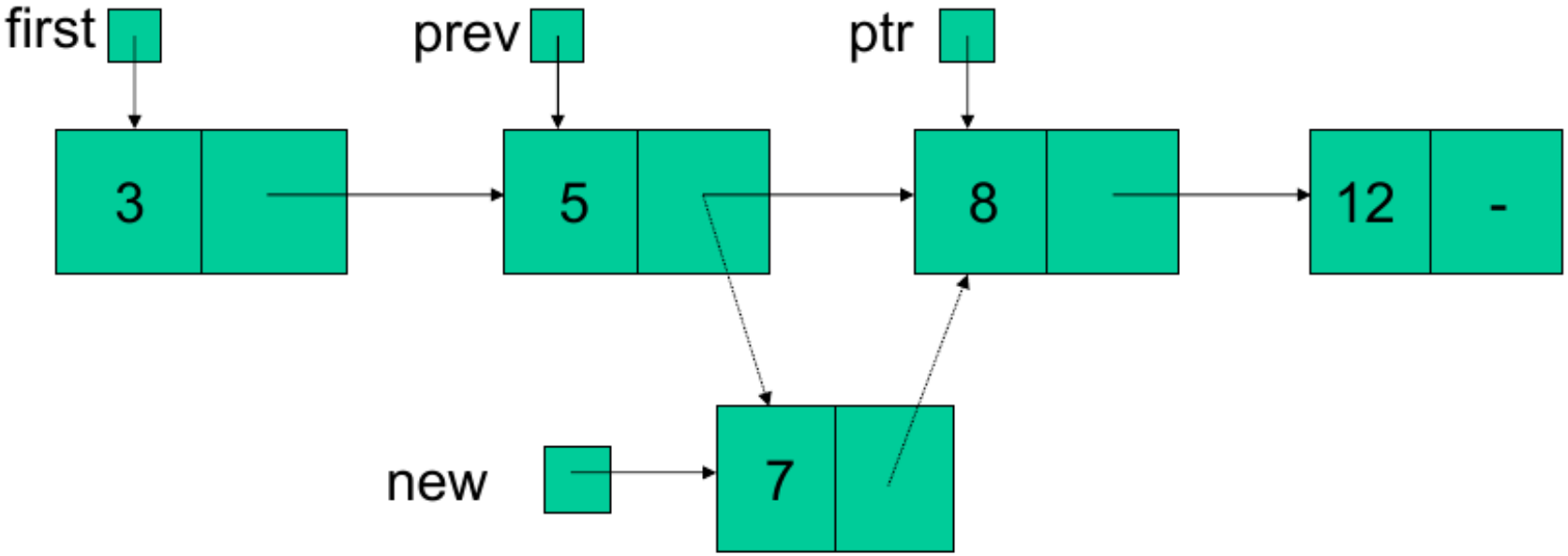
- Removing at the tail of a singly linked list cannot be efficient!
- There is no constant-time way to update the tail to point to the previous node



Insertion

- To insert a data item into an ordered linked list involves:
  - creating a new node containing the data,
  - finding the correct place in the list, and
  - linking in the new node at this place.

Example of an Insertion



- Create new node for the 7
- Find correct place - when ptr finds the 8 (7 < 8)
- Link in new node with previous (even if last) and ptr nodes
- Also check insertion before first node!

Creating a node function

```
Listpointer create_node(int data)
{
  LINK new;
  new = (LINK) malloc (sizeof (ELEMENT));
  new -> data = data;
  return (new);
}
```

Insert function

```
LINK insert (int data, LINK ptr)
{
  LINK new, prev, first;
  new = create_node(data);
  if (ptr == NULL || data < ptr -> value)
  {
    // insert as new first node
    new -> next = ptr;
    // return pointer to first node
    return new;
  }
```

```
else  // not first one ?
  {
    first = ptr; // remember start
    prev = ptr;
    ptr = ptr -> next; // second
    while (ptr != NULL && data > ptr -> data)
    {
      // move along
      prev = ptr;
      ptr = ptr -> next;
    }
    prev -> next = new; // link in
    new -> next = ptr; //new node
    return first;
  }
 // end else
// end insert
}
```