Pointers

EECS 2031

Song Wang
wangsong@eecs.yorku.ca
eecs.yorku.ca/~wangsong/

Acknowledgement

- Some of the covered materials are based on previous EECS2031 offerings:
  - Uyen Trang (UT) Nguyen, Pooja Vashisth, Hui Wang, Manos Papagelis

Using the malloc'd Array

- Once the memory is allocated, it can be used with pointers, or with array notation
- Example:

```
int *p, n, i;
scanf("%d", &n);
p = (int *) malloc (n * sizeof(int));
for (i=0; i<n; ++i)
    scanf("%d", &p[i]);
```

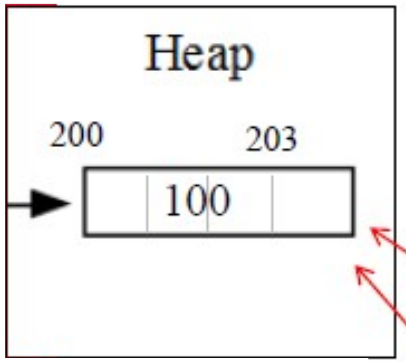
The n integers allocated can be accessed as *p, *(p+1), *(p+2),..., *(p+n-1) or just as p[0], p[1], p[2], ...,p[n-1]

Example

```
int main()
{
  int i,N;
  float *marks;
  float sum=0,avg;

  printf("Input no. of students\n");
  scanf("%d", &N);

  marks = (float *)
       malloc(N * sizeof(float));
```

```
printf("Input marks for %d
students \n",N);
  for (i=0; i<N; i++)
   scanf ("%f", &marks[i]);

  for(i=0;i<N;i++)
    sum += marks[i];

  avg = sum / (float) N;

  printf("Average marks = %f \n",
                avg);
  free (marks);
  return 0;
}
```

```c
#include <stdio.h>

void setArr (int);

int * arr[10]; // global, array of 10 int pointers

int main(int argc, char *argv[])
{

   setArr(1);


   printf("arr [%d] = %d\n", 1, *arr[1]);      // 100

   return 0;
}

/* set arr[index], which is a pointer,
to point to an integer of value 100 */
void setArr (int index){

  int *p = malloc(sizeof(int));
  *p = 100;
  arr[index] = p; // points to heap space
}
```



Debugging

- gdb and gcc with option -g.
- valgrind: illegal accesses, unintialized values, etc.
- Hard to reason about a such complicated program only using these generic tools.
- Use your heap checker to print out more information before it crash and burns!

valgrind

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int *b = calloc(4,  sizeof(int));
  printf("%d\n", b[4]);
  free(b);
  return 0;
}
```

valgrind

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int *b = calloc(4,  sizeof(int));
  printf("%d\n", b[4]);
  free(b);
  return 0;
}
```

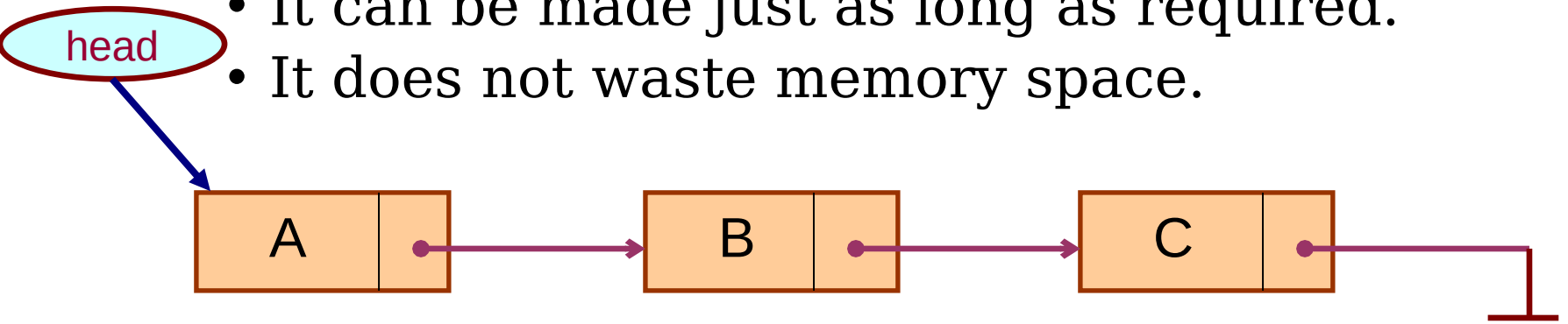
```
==5539== Command: ./e1
==5539==
==5539== Invalid read of size 4
==5539==    at 0x1086FD: main (e1.c:7)
==5539==  Address 0x522d050 is 0 bytes after a block of
size 16 alloc'd
==5539==    at 0x4C31B25: calloc (in
/usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==5539==    by 0x1086F0: main (e1.c:5)
==5539==
==5539== ERROR SUMMARY: 1 errors from 1 contexts
(suppressed: 0 from 0)
```

Linked Lists

- A **linear** collection of self-referential objects, typically called nodes, connected by other links
  - **linear**: for every node in the list, there is one and only one node that precedes it (except for possibly the first node, which may have no predecessor,) and there is one and only one node that succeeds it, (except for possibly the last node, which may have no successor)
  - **self-referential**: a node that has the ability to refer to another node of the same type, or even to refer to itself
  - **node**: contains data of any type, including a reference to another node of the same data type, or to nodes of different data types
  - Usually a list will have a beginning and an end; the

Linked Characteristics

- A linked list is a data structure which can change during execution.
  - Successive elements are connected by pointers.
  - Last element points to NULL.
  - It can grow or shrink in size during execution of a program.
  - It can be made just as long as required.
  - It does not waste memory space.



Linked Characteristics

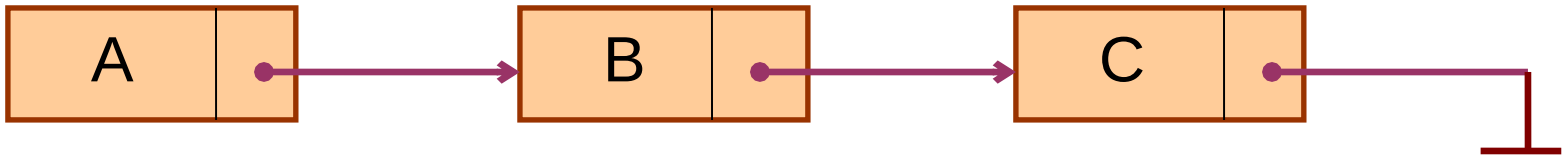
- Keeping track of a linked list:
  - Must know the pointer to the first element of the list (called *start*, *head*, etc.).
- Linked lists provide flexibility in allowing the items to be rearranged efficiently.
  - Insert an element
  - Delete an element

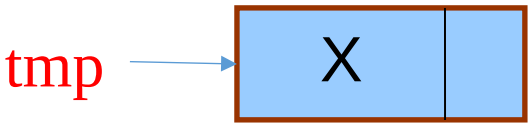
Array versus Linked Lists

- Arrays are suitable for:
  - Inserting/deleting an element at the end.
  - Randomly accessing any element.
  - Searching the list for a particular value.

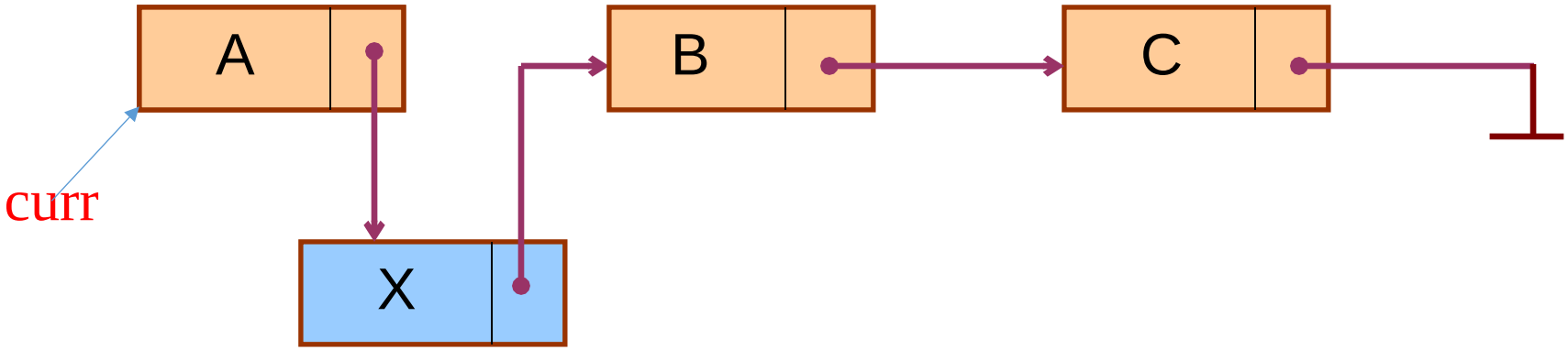
- Linked lists are suitable for:
  - Inserting an element randomly.
  - Deleting an element randomly.
  - Applications where sequential access is required.
  - In situations where the number of elements cannot be predicted beforehand.

Illustration: Insertion





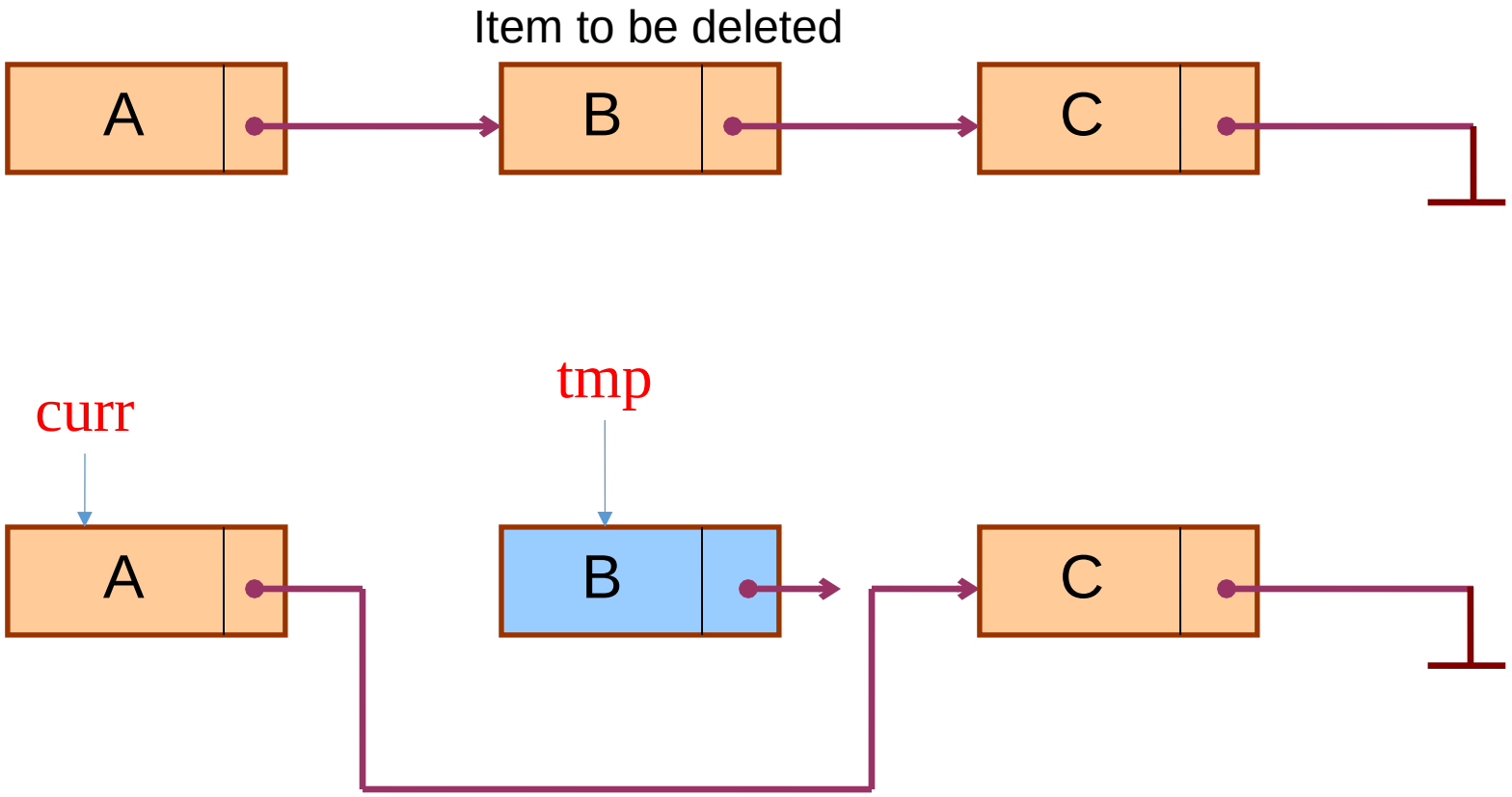
Item to be inserted



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;
}
```

Illustration: Deletion

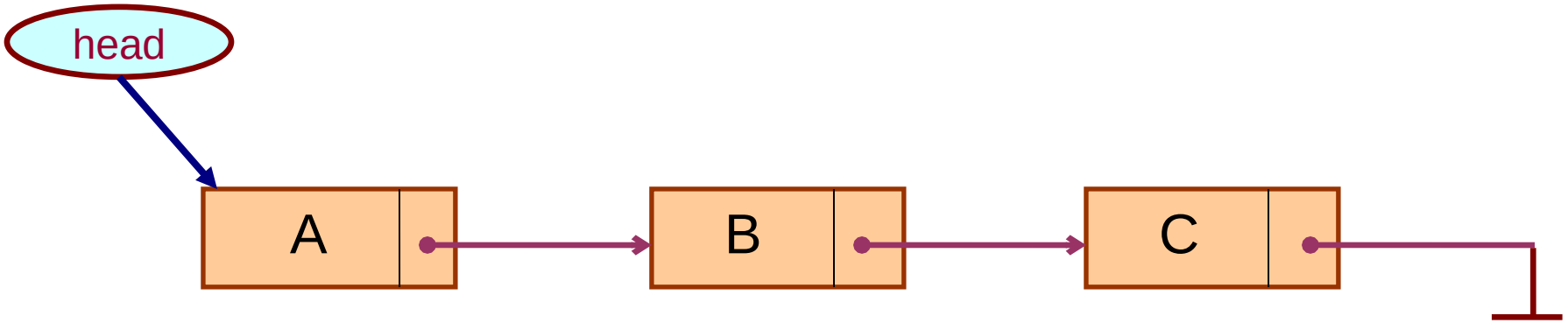


Pseudo-code for deletion

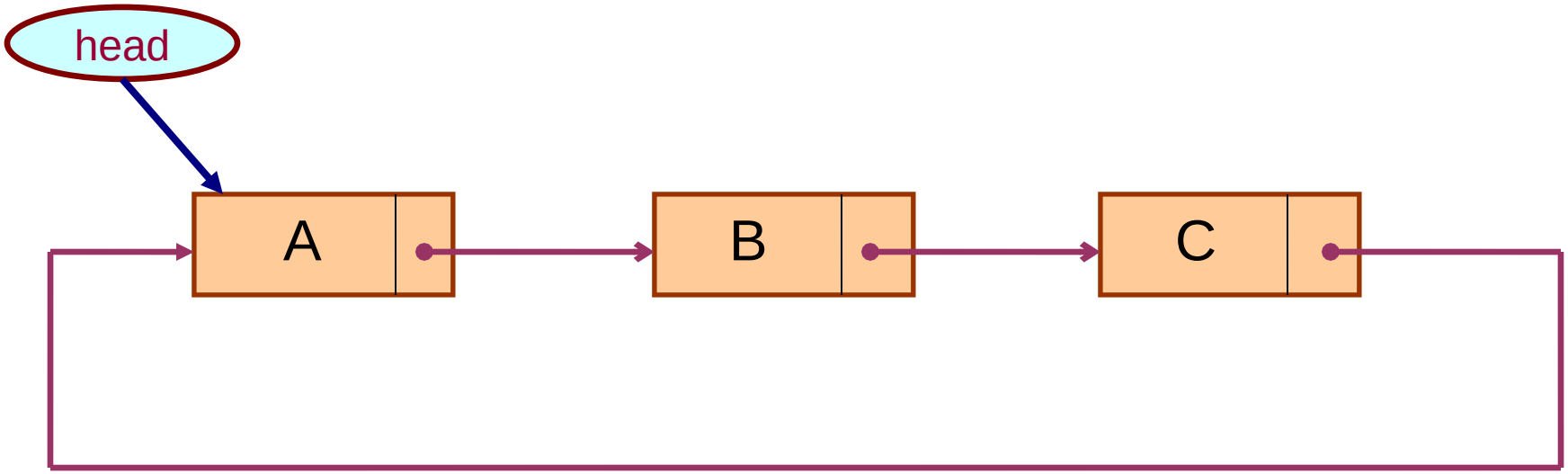
```
typedef struct nd {
   struct item data;
   struct nd * next;
   } node;

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

Types of Lists

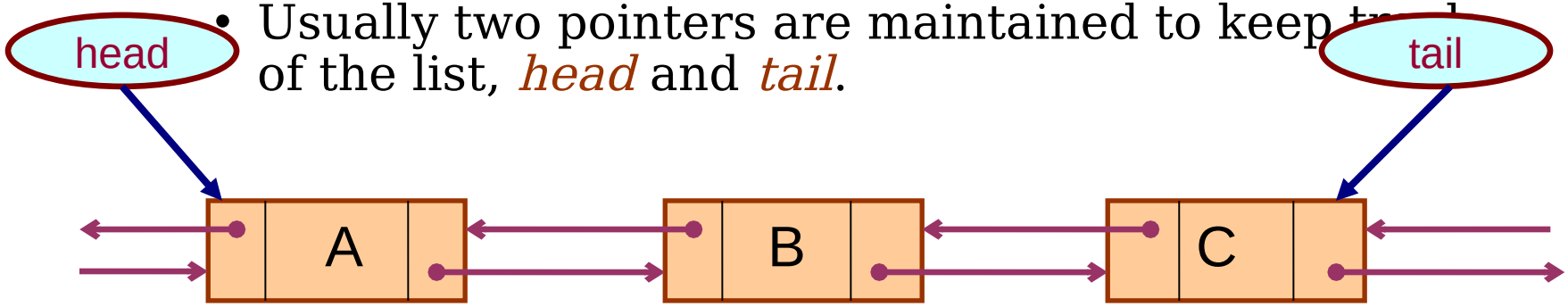
- Depending on how the links are used to maintain adjacent nodes, several different types of linked lists are possible.
  - Linear singly-linked list (or simply linear list)
    - One we have discussed so far.



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



- Double 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*.




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

Basic Operations on a List

- Creating a list
- Traversing the list
- Inserting an item in the list
- Deleting an item from the list
- Concatenating two lists into one

Example: Working with linked list

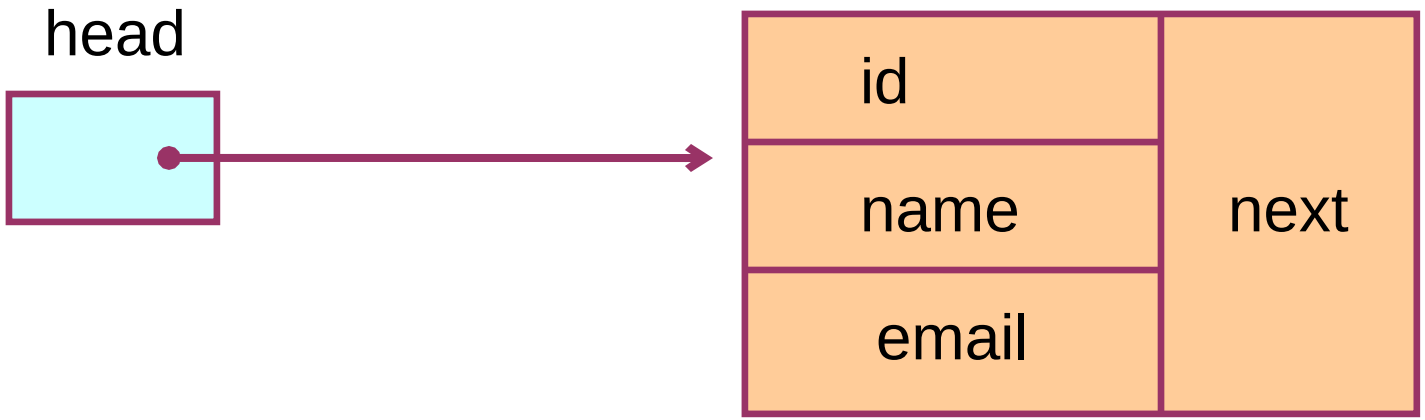
- Consider the structure of a node as follows:

```
struct stud {
              int   id;
              char  name[25];
              char   email[50];
              struct stud *next;
            };

  /* A user-defined data type called “node” */
typedef struct stud node;
node *head;
```

Creating a List

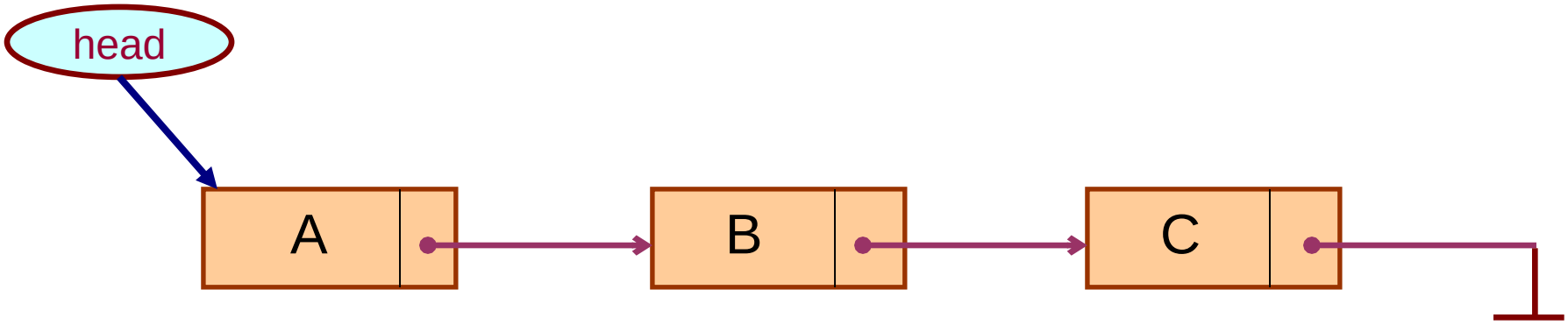
- To start with, we have to create a node (the first node), and make head point to it.

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



Contd.

- 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.



```
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 %s", &p->id, p->name, &p->email);
    }

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

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

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

Traversing the List

- Once the linked list has been constructed and *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.

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

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

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

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

Inserting a Node in a 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 id.
- Convention followed:
  - If the value of id is given as *negative*, the node will be inserted at the *end* of the list.

How to insert?

- 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.

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

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

    printf ("\nData to be inserted: ");
      scanf ("%d %s %s", &new->id, new->name, &new->email);
    printf ("\nInsert before id:");
      scanf ("%d", &id);

    p = *head;

    if (p->id == id)      /* At the beginning */
    {
        new->next = p;
        *head = new;
    }
```

```
  else
     {
        while ((p != NULL) && (p->id != id))

          {
              q = p;
              p = p->next;
          }

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

The pointers q and p always point to consecutive nodes.

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

```
        node *head;
head = create_list();
display (head);
insert (&head);
```

Deleting a node from the list

- Here also we are required to delete a specified node.
  - Say, the node whose `id` field is given.

- Here three scenarios arise:
  - Deleting the first node.
  - Deleting the last node.
  - Deleting an intermediate node.

```
void  delete (node **head)
{
    int  id; /* To be deleted */
    node  *p, *q;

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

    p = *head;
    if  (p->id == id)
              /* delete the first element */
    {
        *head = p->next;
        free (p);
    }
```

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

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

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