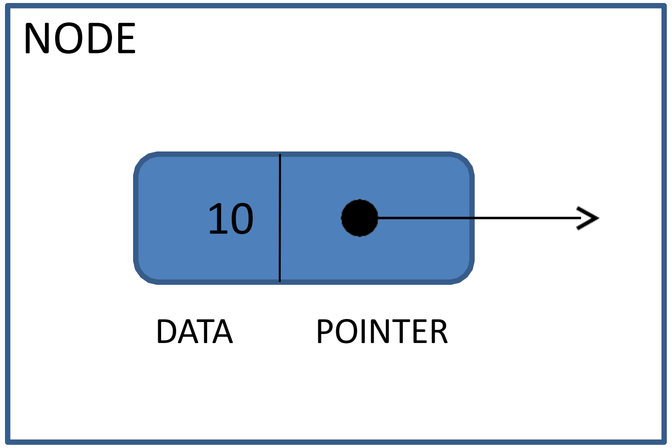
Linked List

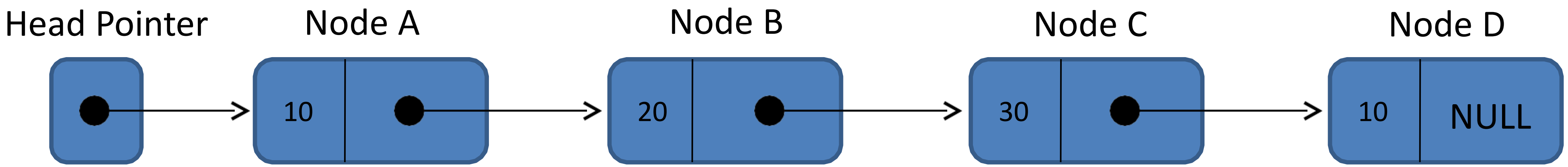
What is Linked List

- A *linked list* is a linear collection of data elements, **called nodes**, each pointing to the next node by means of a pointer.
- It is a data structure consisting of a group of nodes which together represent a sequence.
- Each node stores
  - Data field
  - Address field (reference to the next node)



Linked List

- A list implemented by each item having a link to the next item.
- Head points to the first node.
- Last node points to NULL.



A graphical view of a linked list

Why do we use Linked List?

Linked lists are preferable over arrays when:

A. We don't know how many items will be in the list. With arrays, you may need to re-declare and copy memory if the array grows too big

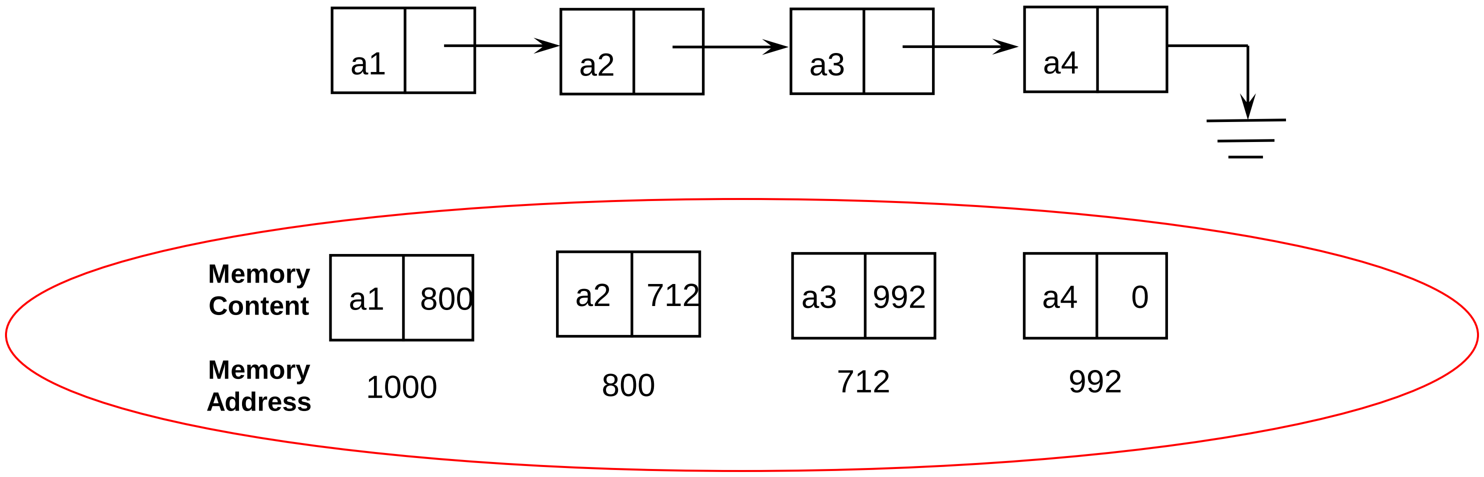
B. We don't need random access to any elements

C. We want to be able to insert items in the middle of the list (such as a priority queue)

Singly-linked lists vs. 1D-arrays

ID-array	Singly-linked list
Fixed size: Resizing is expensive	Dynamic size
Insertions and Deletions are inefficient: Elements are usually shifted	Insertions and Deletions are efficient: No shifting
Random access i.e., efficient indexing	No random access → Not suitable for operations requiring accessing elements by index such as sorting
No memory waste if the array is full or almost full; otherwise may result in much memory waste.	Extra storage needed for references; however uses exactly as much memory as it needs
Sequential access is faster because of greater locality of references [Reason: Elements in contiguous memory locations]	Sequential access is slow because of low locality of references [Reason: Elements not in contiguous memory locations]

What does the memory look like?



Types of Linked List

There are three types of linked list as given below:

1. Singly Linked List
2. Doubly Linked List
3. Circular Linked List

Singly Linked List Operations

There are several operations in singly linked list:

1. Traverse
2. Insertion
3. Deletion
4. Searching

Nodes

- Nodes are the units which makes up linked list.
- In singly linked list, nodes are actually structures made up of data and a pointer to another node.
- Usually we denote the pointer as “next”

*Nodes structure

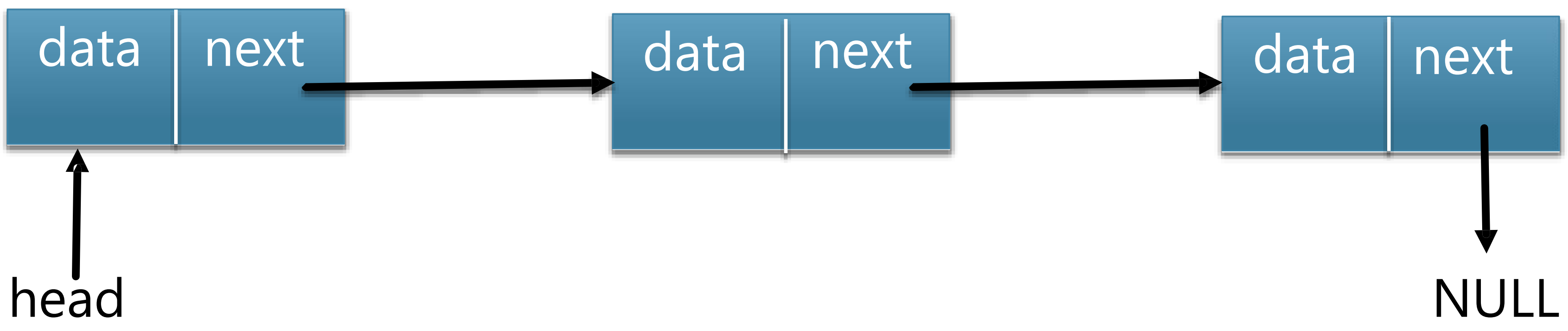
```
struct node
{
    int data;                    //This is where we will store data
    struct node *next;           //This is the link part
};

struct node *head=NULL;
/*Initially we assign "NULL" to the head for later operational purpose*/
```

Creation

- Generally, we use dynamic memory allocation to create our desired number of nodes for linked list in program run time.

*Visual Representation of Linked List

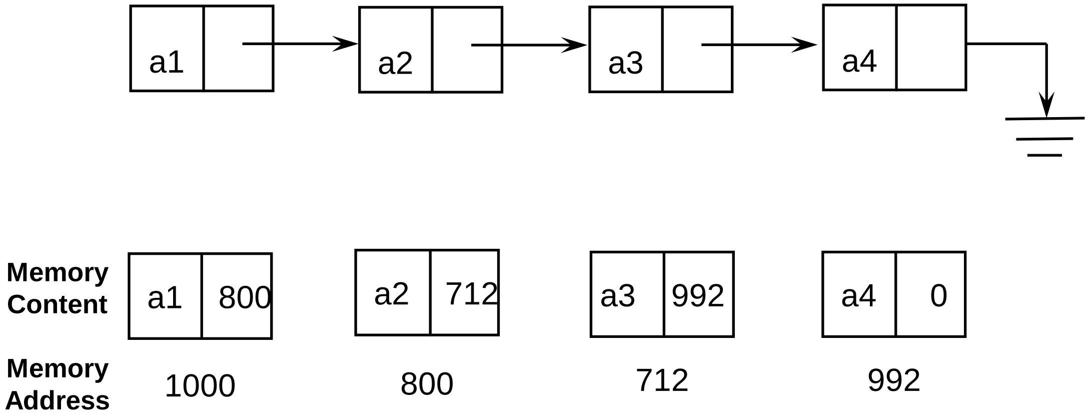


Traverse

- To traverse the linked list we need to follow the steps given below:
  - ➢ We need a pointer which is assigned with the address of head pointer, and another pointer which is assigned to the current location.
  - ➢ Then we start a loop until the last node of the linked list.

```
traverse():
Begin
        cur = head
        forever:
                if cur ->next == NULL
                        break
                cur = cur->next
End
```

Traverse




```
traverse():
Begin
        cur = head
        forever:
                if cur ->next == NULL
                        break
                cur = cur->next
End
```

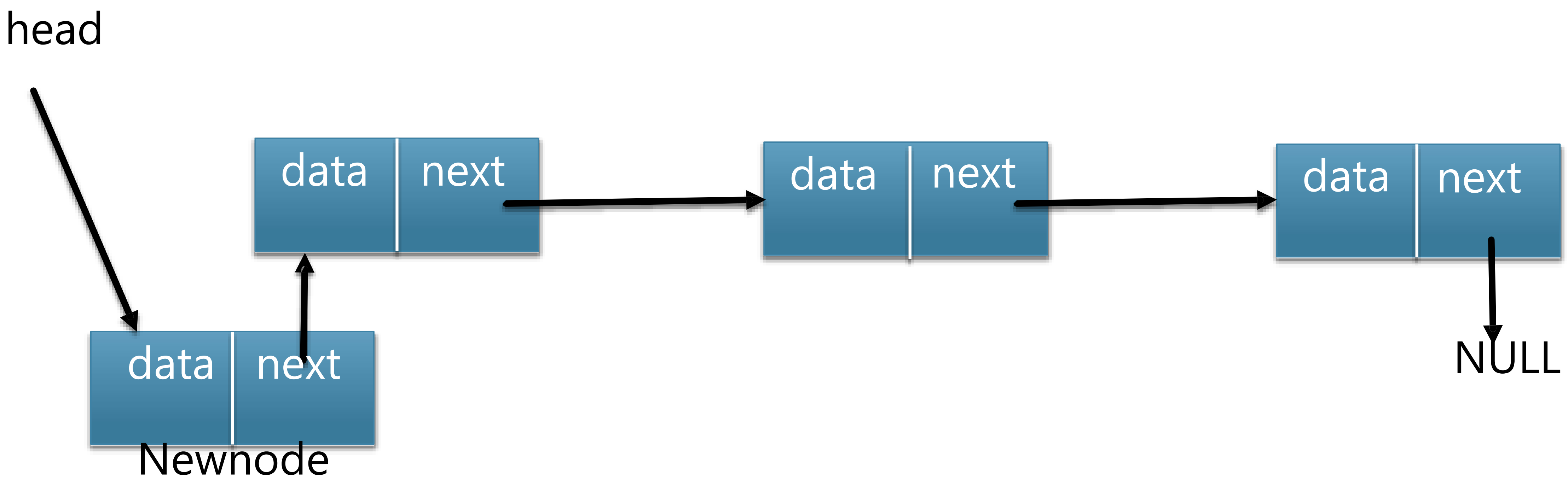
Insertion

- Insertion operation is used to insert a new node in the linked list.
- Insertion is of three types. They are:
  - Inserting at first
  - Inserting at last
  - Inserting at mid

Insertion at first (Prepend)

- There are two steps to be followed to insert a node at first position. They are:
  - ➢ Make the next pointer of the new node point towards the first node of the list.
  - ➢ Make the head pointer point towards this new node.

*Visual Representation of Insertion at First



//Inserting at first

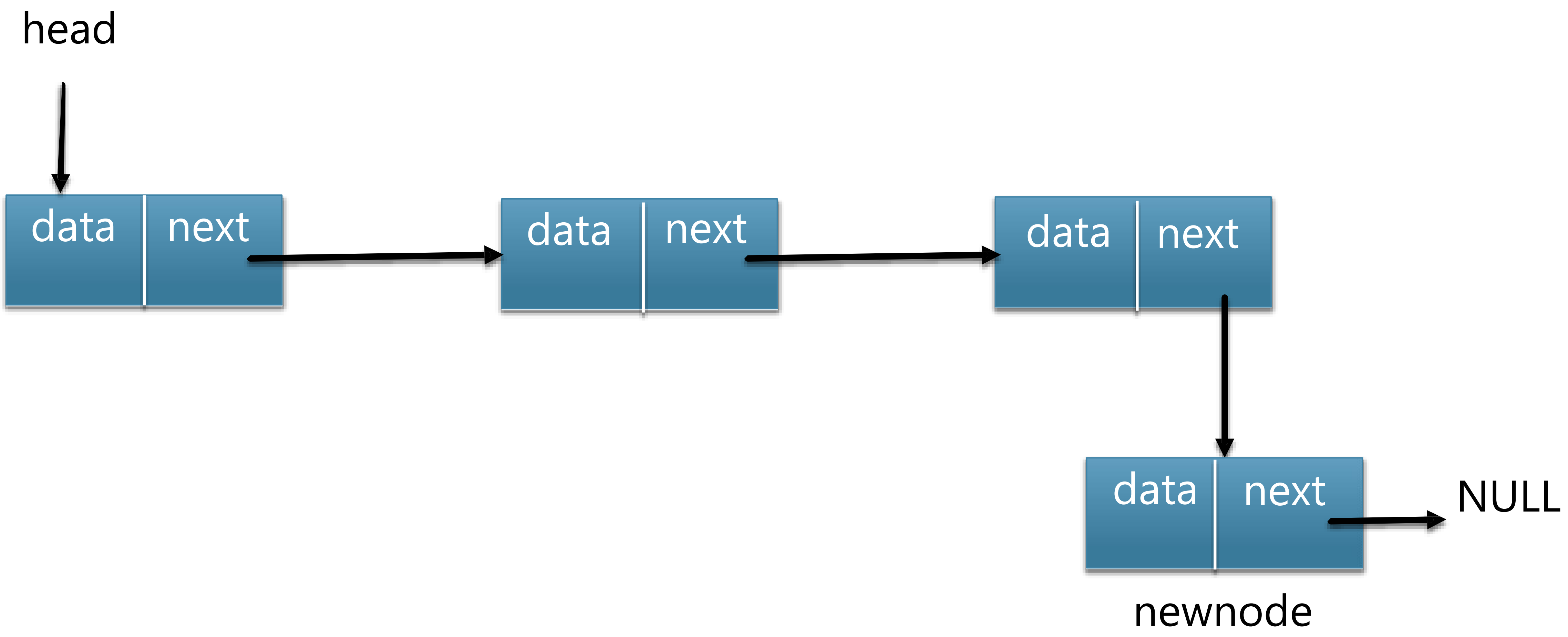
```
insertFirst(data):
    Begin
      create a new node
      newnode -> data = data
      newnode -> next = null

       if head == null
            head = newnode
       else
            newnode->next = head
            head = newnode
    End
```

Insertion at last (Append)

- Inserting at the last position is a simple process.
  - To insert at last we just simply need to make the next pointer of last node to the new node.

*Visual Representation of Insertion at Last



//Inserting at Last

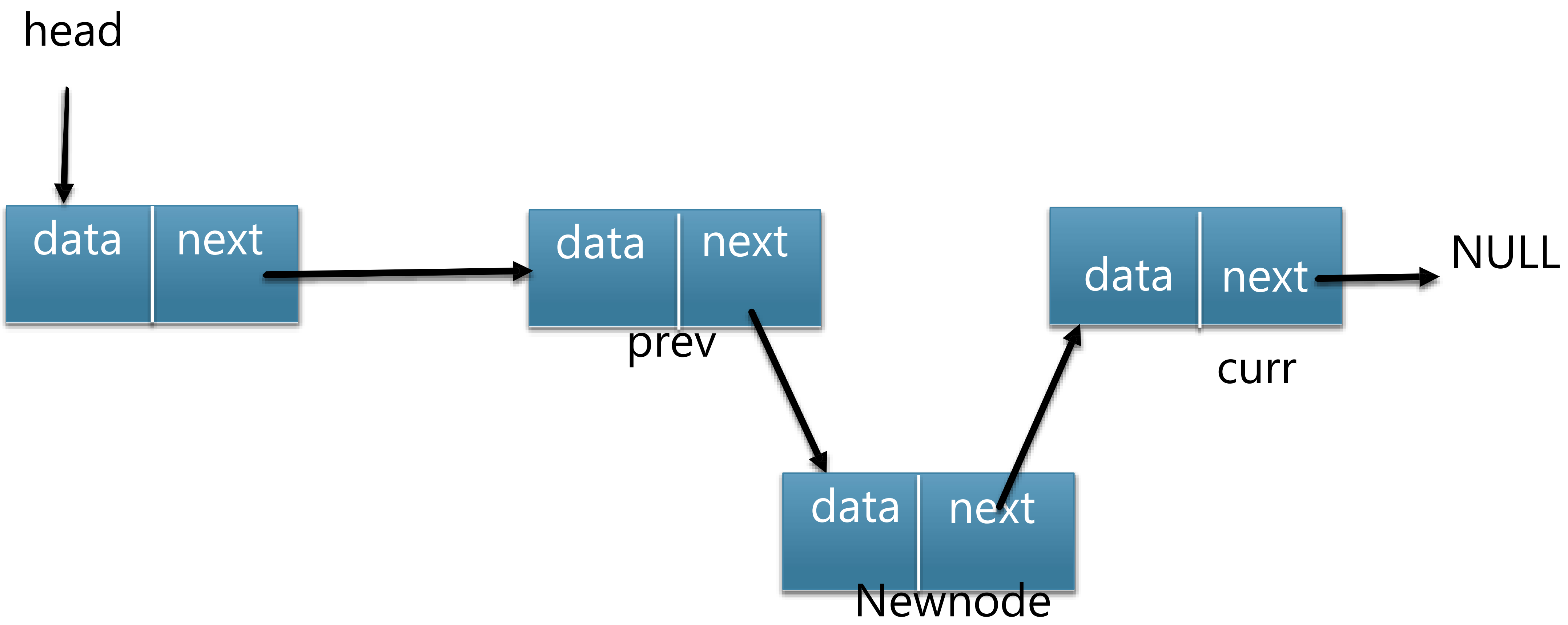
```
insertLast(data):
      Begin
        create a new node
        newnode -> data = data
        newnode -> next = null

        if the list is empty, then
             head = newnode
        else
            curr = head
             forever:
                     if curr -> next == null
                         break
                     curr= curr->Next
             curr - > next = newnode
      End
```

Insertion at position

- There are several steps to be followed to insert a node at specific position. They are:
  - Traverse the list until we reach the position where the new node is going to be inserted.
  - Assign the location of the previous Node of the position to pointer named 'prev' and the position to a pointer named 'curr'.
  - Make the next pointer of 'prev' to point newnode, and the next pointer of newnode to 'curr'.

*Visual Representation of Insertion at Position



//Inserting at position

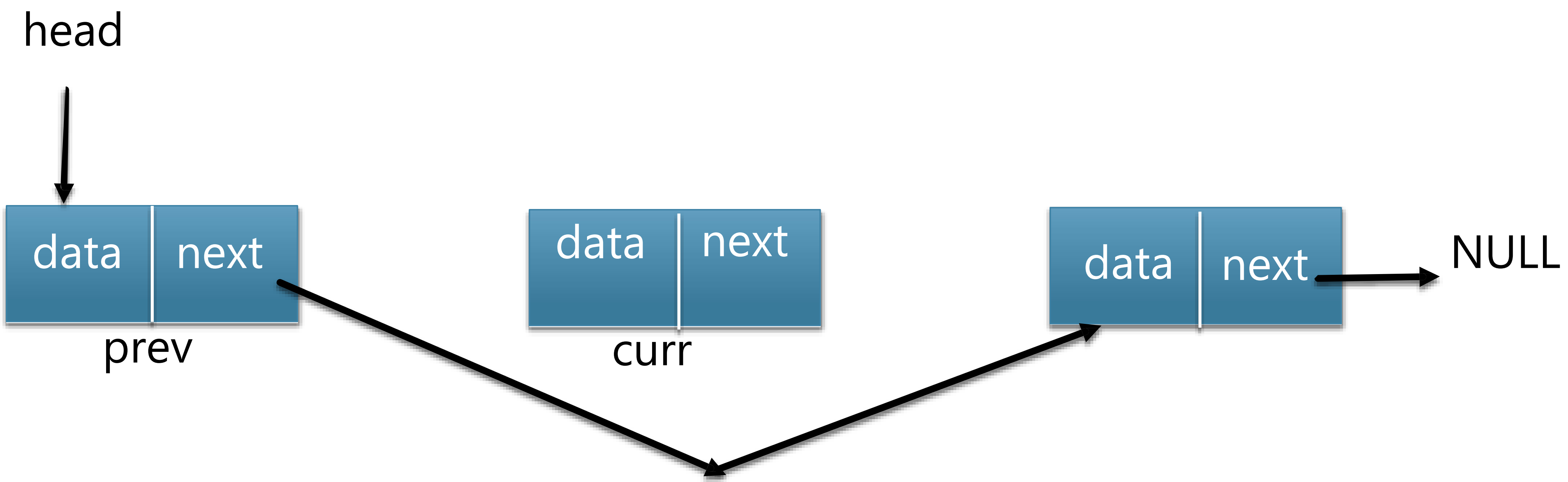
```
insertAtPos(data, pos):
Begin
        create a new node
        newnode -> data = data
        newnode -> next = null

        curPos = 1
        if the list is empty, then
             head = newnode
        else
            curr = head
            forever:
                  if curPos == pos || curr -> next == null
                      break
                  prev = curr
                  curr= curr->Next
                  curPos++
            prev -> next = newnode
            newnode -> next = curr
End
```

Deletion

- Deleting is a simple process.
  - At first we need to find the previous and the next node of that particular node.
  - Then we need to point the next pointer of the previous node to the next node of the particular node we want to delete.

*Visual Representation of Deletion



//Delete

```
deleteAtPos(pos):
        Begin
                curr = head
                curPos = 1
                   if head is null, then
                        it is Underflow and return
                  else if pos ==1
                         head = curr>Next
                         delete curr
                   else
                          forever:
                                if curPos == pos || curr -> next == null
                                     break
                               prev = curr
                               curr= curr->Next
                               curPos++
                          prev->Next = curr->Next
                         Delete curr
        End
```

Searching

- To search in a linked list we need to follow the steps given below:
  - We need a pointer which is assigned with the address of head pointer.
  - Then we start a loop until the last node of the linked list to search.

//Searching

```
search(value):
Begin
        pos = 1
        cur = head
        forever:
                if cur ->next == null
                        break
                cur = cur->next
                if curr -> data == value
                        return pos
                pos++
End
```

Uses of Linked List

1. Web-browsers

A good example is web-browsers, where it creates a Linked history (traversal of a list) or press back button, the previous node's data is fetched.

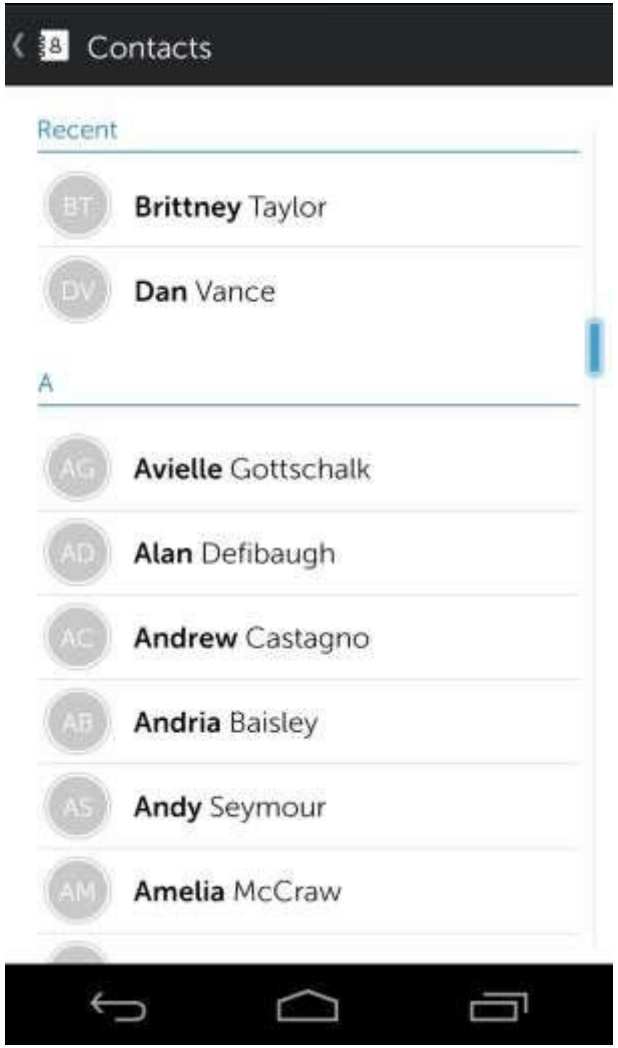
2. Stacks and queues

It is used to develop stacks and queues which have lots of applications.

3. Image viewer

4. Phonebook

In phonebook the names are sorting in ascending order. If we want to add a new contact then the memory for the new contact is created by linked list.



Advantages

- Linked lists are a dynamic data structure, allocating the needed memory while the program is running.
- Insertion and deletion node operations are easily implemented in a linked list.
- Linear data structures such as stacks and queues are easily executed with a linked list.
- They can reduce access time and may expand in real time without memory overhead.

Disadvantages

- They have a quite difficult to use more memory due to pointers requiring extra storage space.
- Nodes in a linked list must be read in order from the beginning as linked lists are inherently sequential access.
- Nodes are stored in continuously, greatly increasing the time required to access individual elements within the list.
- Difficulties arise in linked lists when it comes to reverse traversing. For instance, singly linked lists are cumbersome to navigate backwards and while doubly linked lists are somewhat easier to read, memory is wasted in allocating space for a back pointer.