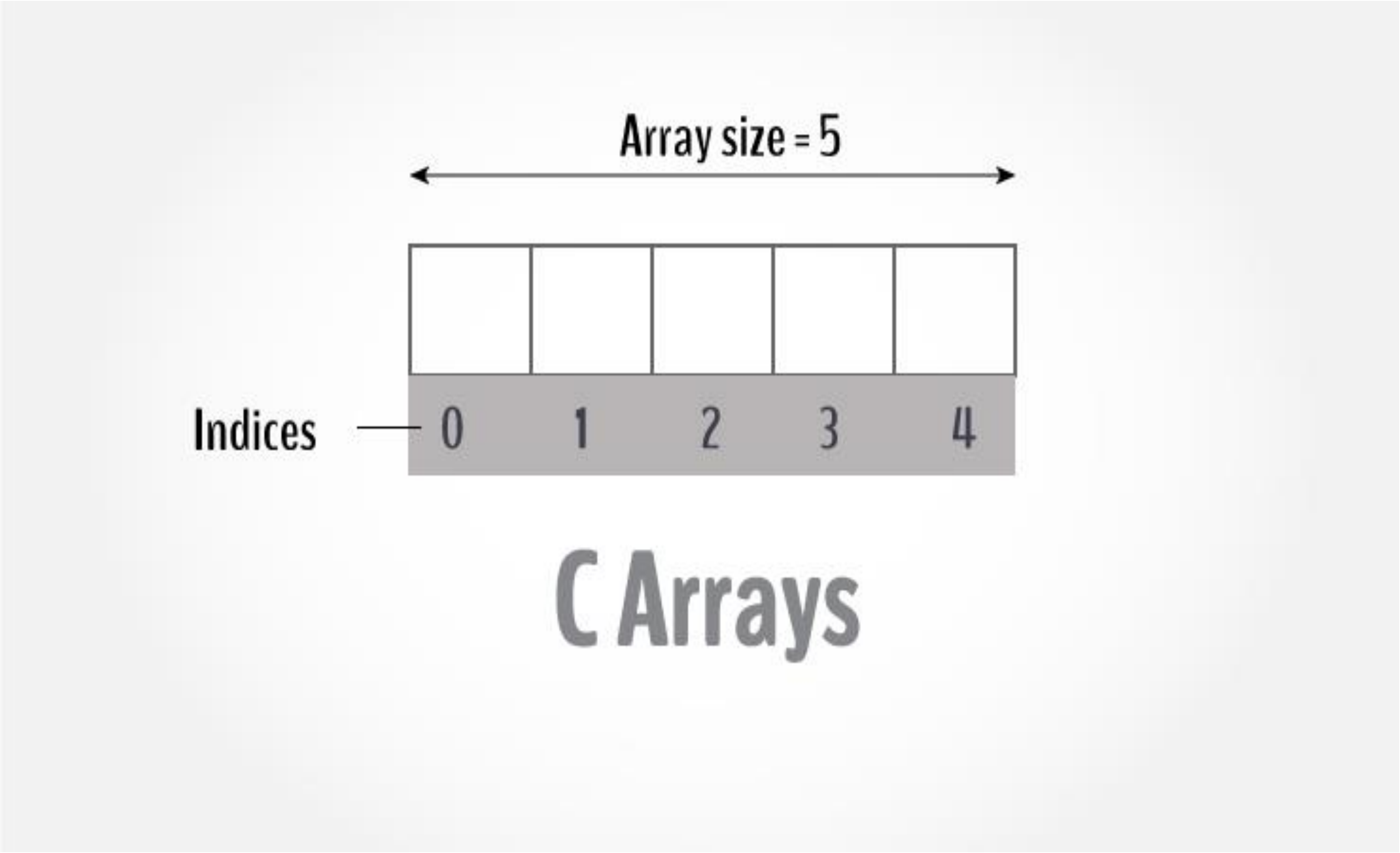
Arrays

- An Array consists of Collection of elements in same data type.
- Elements are stored in consecutive memory locations.
- An array is a set of pairs, index and value. For each index which is defined, there is a value associated with that index.

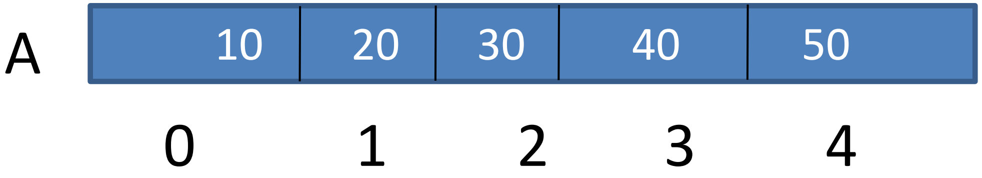

Array size = 5
Indices
0
1
2
3
4
C Arrays

Array

```
Structure ARRAY(value, index)
    Declare CREATE()->array
            RETRIEVE(array,index)->value
            STORE(array,index,value)->array;
    for all Aε array, i,j ε index,, x ε value let
RETRIVE(CREATE,i)::=error
RETRIVE(STORE(A,i,x),j)::=
If EQUAL(i,j) then x else RETRIVE(A,J)
End
End ARRAY
```

- The function CREATE()
  - It produces a new empty array.

- RETRIVE(array,index)->value
  - It takes input an array and an index, and either returns the appropriate value or an error.



- STORE(array,index,value)->array
  - It is used to enter new index pairs.

ORDERED LISTS

- Collection of elements are arranged in sequential order.
- Example
  - Days of the week
  - Values in a card deck
  - (a1,a2,a3….,an)

Operations

- Find the length of the list, n.
- Read the list from left to right.
- Retrieve the i-th element from the list.
- Store a new value into the i-th positon
- Insert a new element at position i.
- Delete the element at position i.

LIST ADT

- A sequence of zero or more elements

  $A_1, A_2, A_3, \ldots A_N$
- N: length of the list
- A_1: first element
- A_N: last element
- A_i: position i
- If N=0, then empty list
- Linearly ordered
  - A_i precedes A_{i+1}
  - A_i follows A_{i-1}

Operations

- printList: print the list
- makeEmpty: create an empty list
- find: locate the position of an object in a list
  - list: 34,12, 52, 16, 12
  - find(52) → 3
- insert: insert an object to a list
  - insert(x,3) → 34, 12, 52, x, 16, 12
- remove: delete an element from the list
  - remove(52) → 34, 12, x, 16, 12
- findKth: retrieve the element at a certain position

Implementation of ADT

- Choose a **data structure** to represent the ADT
  - E.g. arrays, records, etc.
- Each operation associated with the ADT is implemented by one or more subroutines
- Two standard implementations for the list ADT
  - Array-based
  - Linked list

Array Implementation

- Requires an estimate of the maximum size of the list
  - ➢ waste space
- printList and find: linear
- findKth: constant
- insert and delete: slow
  - e.g. insert at position 0 (making a new element)
    - requires first pushing the entire array down one spot to make room
  - e.g. delete at position 0
    - requires shifting all the elements in the list up one
  - On average, half of the lists needs to be moved for either operation

Pointer implementation(Linked List)

- Ensure that the list is not stored contiguously
  - use a linked list
  - a series of structures that are not necessarily adjacent in memory

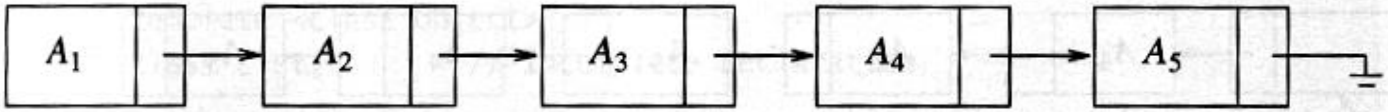
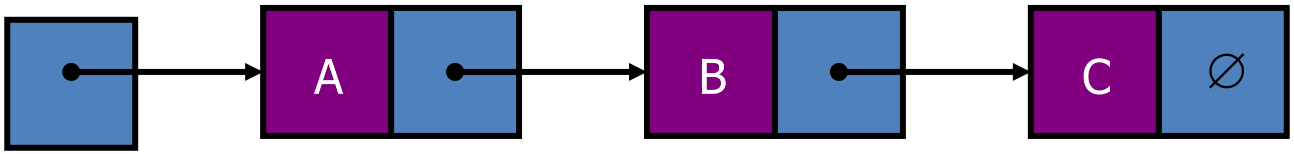


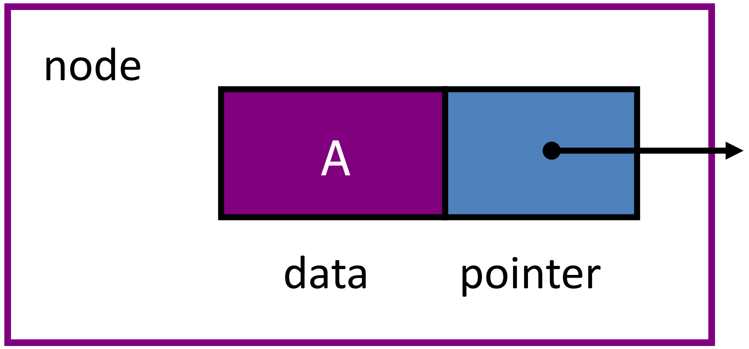
Figure 3.1 A linked list

- Each node contains the element and a pointer to a structure containing its successor
  - the last cell's next link points to NULL
- Compared to the array implementation,
  - ✓the pointer implementation uses only as much space as is needed for the elements currently on the list
  - ☐but requires space for the pointers in each cell

Linked List



- A *linked list* is a series of connected *nodes*
- Each node contains at least
  - A piece of data (any type)
  - Pointer to the next node in the list
- *Head*: pointer to the first node
- The last node points to `NULL`



A Simple Linked List Class

- We use two classes: **`Node`** and **`List`**
- Declare `Node` class for the nodes
  - `data`: `double`-type data in this example
  - `next`: a pointer to the next node in the list

```
class Node {
public:
        double data;              // data
        Node*           next;              // pointer to next
};
```

A Simple Linked List Class

- Declare `List`, which contains
  - `head`: a pointer to the first node in the list.

    Since the list is empty initially, `head` is set to `NULL`
  - Operations on `List`

```
class List {
public:
      List(void) { head = NULL; }        // constructor
      ~List(void);                       // destructor

      bool IsEmpty() { return head == NULL; }
      Node* InsertNode(int index, double x);
      int FindNode(double x);
      int DeleteNode(double x);
      void DisplayList(void);
private:
      Node* head;
};
```

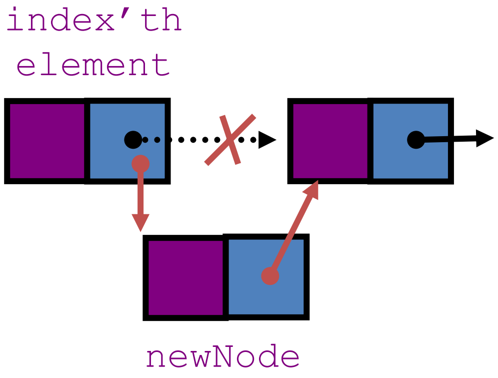
Operations of `List`

- `IsEmpty`: determine whether or not the list is empty
- `InsertNode`: insert a new node at a particular position
- `FindNode`: find a node with a given value
- `DeleteNode`: delete a node with a given value
- `DisplayList`: print all the nodes in the list

Inserting a new node

- `Node* InsertNode(int index, double x)`
    - Insert a node with data equal to `x` after the `index’th` elements. (i.e., when `index = 0`, insert the node as the first element; when `index = 1`, insert the node after the first element, and so on)
    - If the insertion is successful, return the inserted node. Otherwise, return `NULL`. (If `index` is < 0 or > length of the list, the insertion will fail.)
- Steps

1. Locate `index`’th element
2. Allocate memory for the new node
3. Point the new node to its successor
4. Point the new node’s predecessor to the new node



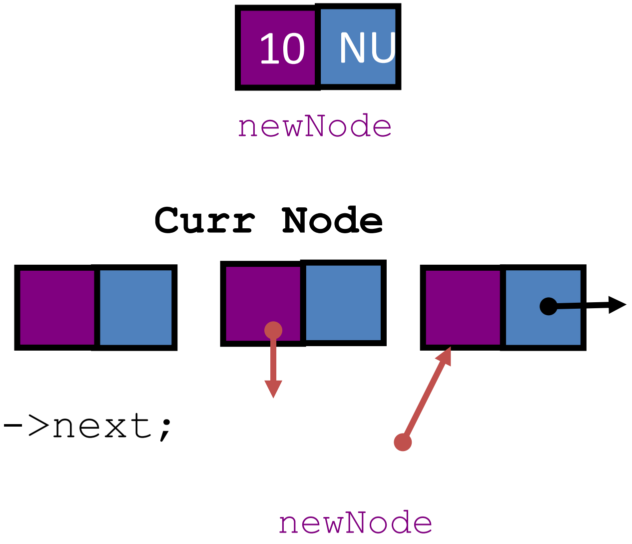
Inserting a new node

- Possible cases of `InsertNode`
  1. Insert into an empty list
  2. Insert in front
  3. Insert at back
  4. Insert in middle
- But, in fact, only need to handle two cases
  - Insert as the first node (Case 1 and Case 2)
  - Insert in the middle or at the end of the list (Case 3 and Case 4)

Inserting a new node

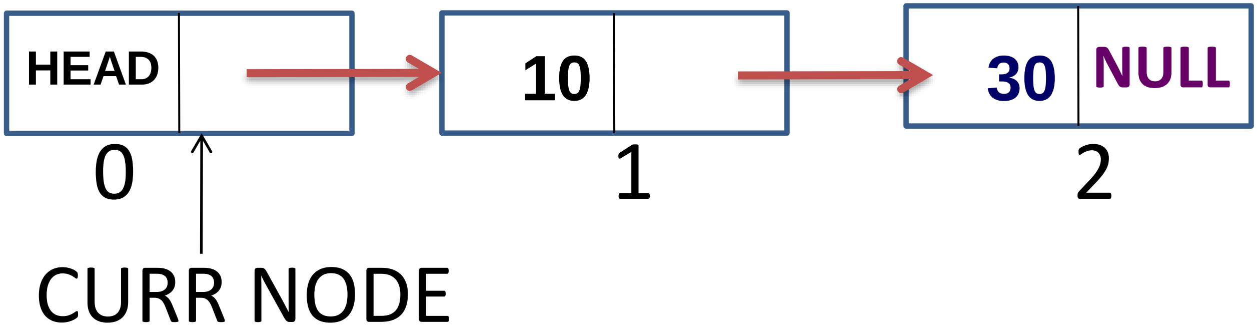
```
List::InsertNode(int index, double x) {
    if (index < 0) return NULL;

    int currIndex  =       1;
    Node* currNode =       head;
    while (currNode && index > currIndex) {
            currNode       =       currNode->next;
            currIndex++;
    }
    if (index > 0 && currNode == NULL) return NULL;

    Node* newNode  =       new     Node;
    newNode->data  =       x;
    if (index == 0) {
            newNode->next  =       head;
            head           =       newNode;
    }
    else {
            newNode->next  =       currNode->next;
            currNode->next =       newNode;
    }
    return newNode;
}
```

Try to locate `index`'th node. If it doesn't exist, return `NULL`.

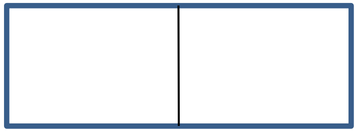


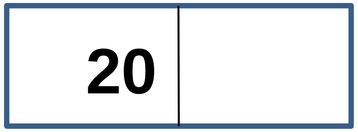
Inserting a new node

Example: InsertNode(int 1,double 10)

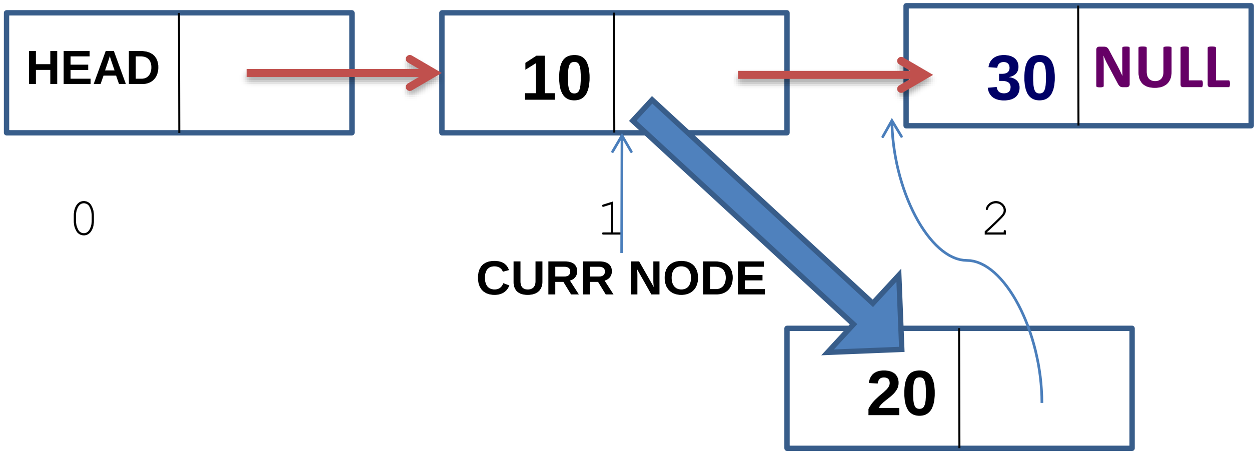


```
Node* newNode = new Node
newNode->data = 20;
```

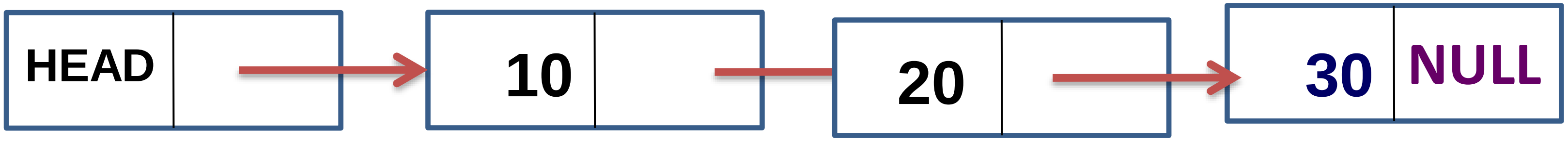




- `newNode->next =    currNode->next;`



- `currNode->next    =    newNode;`



Finding a node

- `int FindNode(double x)`
  - Search for a node with the value equal to `x` in the list.
  - If such a node is found, return its position. Otherwise, return 0.

```
int List::FindNode(double x) {
      Node* currNode      =      head;
      int currIndex =      1;
      while (currNode && currNode->data != x) {
            currNode      =      currNode->next;
            currIndex++;
      }
      if (currNode) return currIndex;
      return 0;
}
```

Deleting a node

- **`int DeleteNode(double x)`**
  - Delete a node with the value equal to `x` from the list.
  - If such a node is found, return its position. Otherwise, return 0.
- Steps
  - Find the desirable node (similar to `FindNode`)
  - Release the memory occupied by the found node
  - Set the pointer of the predecessor of the found node to the successor of the found node
- Like `InsertNode`, there are two special cases
  - Delete first node
  - Delete the node in middle or at the end of the list

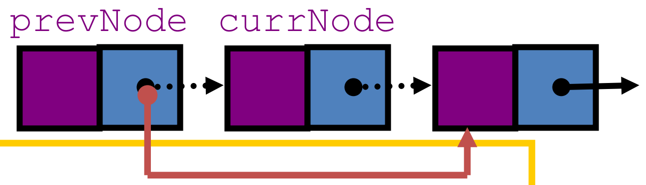
Deleting a node

```
int List::DeleteNode(double x) {
        Node* prevNode  =       NULL;
        Node* currNode  =       head;
        int currIndex   =       1;
        while (currNode && currNode->data != x) {
                prevNode        =       currNode;
                currNode        =       currNode->next;
                currIndex++;
        }
        if (currNode) {
                if (prevNode) {
                        prevNode->next  =       currNode->next;
                        delete currNode;
                }
                else {
                        head            =       currNode->next;
                        delete currNode;
                }
                return currIndex;
        }
        return 0;
}
```

Try to find the node with its value equal to `x`

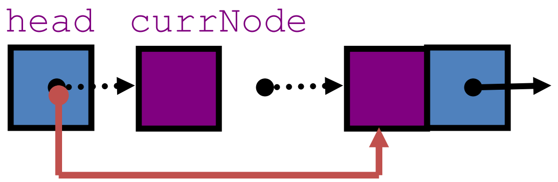
Deleting a node

```
int List::DeleteNode(double x) {
        Node* prevNode =        NULL;
        Node* currNode =        head;
        int currIndex  =        1;
        while (currNode && currNode->data != x) {
                prevNode        =       currNode;
                currNode        =       currNode->next;
                currIndex++;
        }
        if (currNode) {
                if (prevNode) {
                        prevNode->next =        currNode->next;
                        delete currNode;
                }
                else {
                        head            =       currNode->next;
                        delete currNode;
                }
                return currIndex;
        }
        return 0;
}
```



Deletion

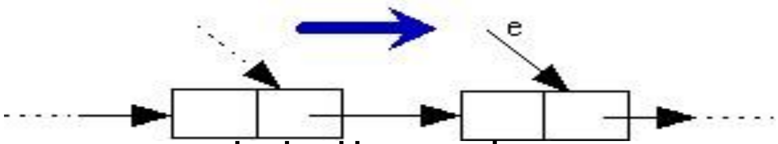
```
int List::DeleteNode(double x) {
        Node* prevNode =      NULL;
        Node* currNode =      head;
        int currIndex  =      1;
        while (currNode && currNode->data != x) {
                prevNode        =      currNode;
                currNode        =      currNode->next;
                currIndex++;
        }
        if (currNode) {
                if (prevNode) {
                        prevNode->next  =      currNode->next;
                        delete currNode;
                }
                else {
                        head            =      currNode->next;
                        delete currNode;
                }
                return currIndex;
        }
        return 0;
}
```



Traversal

Begin at the first node, then follow each *next* reference until the traversal condition is satisfied or until you come to the end.

```
e = e.next;
```



To move an **Element** reference **e** from one node to the next use:

```
public int countNodes(){
    int count = 0;
    Element e = head;
    while(e != null){
        count++;
         e = e.next;
    }
    return count;
}
```

Example: Count the number of nodes in a linked list.

Printing all the elements

- `void DisplayList(void)`
  - Print the data of all the elements
  - Print the number of the nodes

```
void List::DisplayList()
{
   int num           =     0;
   Node* currNode    =     head;
   while (currNode != NULL){
       cout << currNode->data << endl;
       currNode      =     currNode->next;
       num++;
   }
   cout << "Number of nodes in the list: " << num << endl;
}
```