Arrays have the following disadvantages:

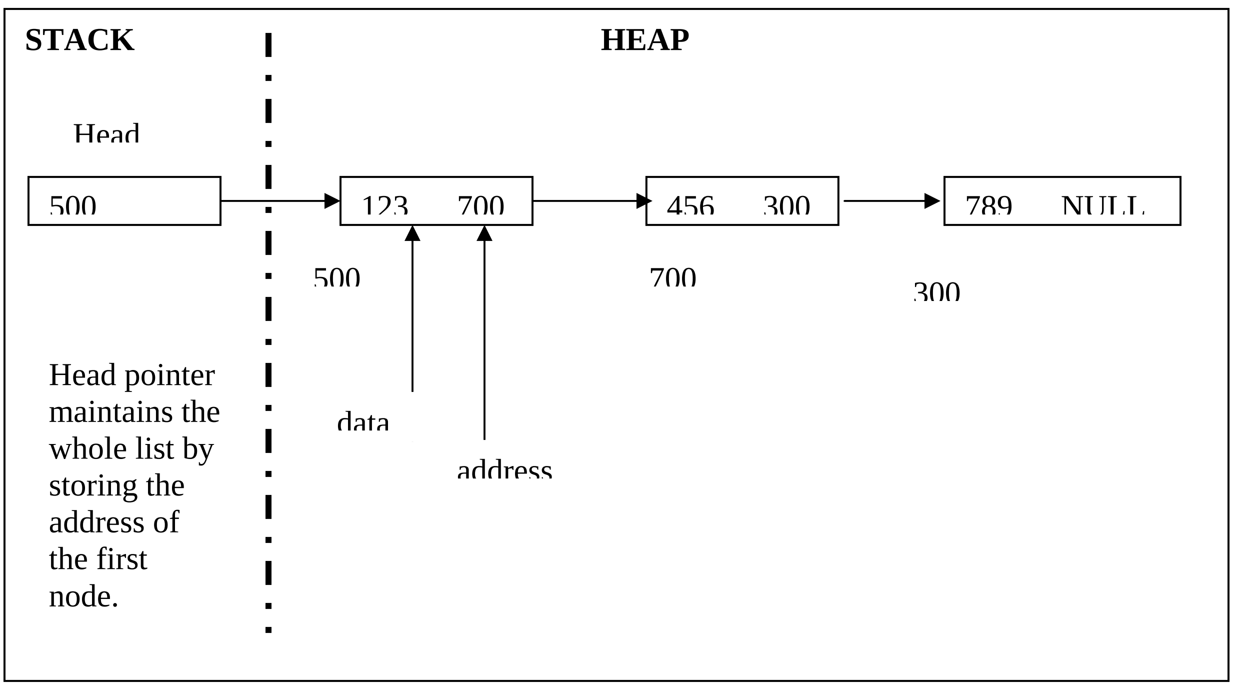
- Fixed size: specified at compile time or deferred until run time. Dynamically resizing an array with realloc( ) requires some real programming effort.(Dynamic Arrays we have discussed)
- Because of fixed size programmers tend to allocate very large arrays and as a result either memory is wasted or the program crashes.
- Inserting new elements at the beginning of the array is expensive, as existing elements need to be shifted
- We have discussed the problem of unordered array. Insertion is easy, searching/deletion is expensive.
- We have discussed the maintenance of orderly array. Insertion is costly, and searching/deletion is a bit easy.

Pointer variables are an extremely useful data type. You can use them to implement the data structure known as a “Linked List”. It has many advantages over arrays.

Linked List

Linked List allocates space for each element separately. These elements are generally referred to as “Nodes”. Pointers link the nodes to each other. Each node contains two fields – data and address.
Data stores whatever the user wants, whereas address stores the pointer to the next node.

Nodes are allocated by using the **`new`**, and needs to be freed explicitly using the **`delete`**. This memory is allocated on the heap and not on the stack.



Class Node

```
template <class T>
class Node {
  // declare class List a friend so that it can access
  //Node's private vars.
   friend class List<T>;
 public:
    // constructors
    Node()  { nextPtr = 0;  }
    Node(const T & d)  { data = d; nextPtr = 0; }
    Node(const T & d, Node<T> *n)
          {data = d; nextPtr = n; }

  private:
    T data;              // data
    Node<T> *nextPtr;    // next node in the list
};
```

Class List

```
// The List will contain nodes linked together by pointers
template <class T>
class List {
public:
   List() { head = 0;  }  // constructor
   List(const List<T> &); // copy constructor
   ~List();     // destructor
   void deleteList();
   void insertAtFront( const T & );
   void removeAtFront();
   void print() const;
   unsigned int getSize() const ;
   List& operator = (const List<T> &);
private:
   Node<T> *head;  // pointer to first node
};
```

Destructor

```
// Destructor
template<class T>
List<T>::~List()
{
   deleteList();

}
```

DeleteList

```
template<class T>
void List<T>::deleteList()
{
   if (head != 0) {    // List is not empty
      cout << "Destroying nodes ...\n";
      Node<T> *currentPtr = head;
      Node<T> *tempPtr;

      while ( currentPtr != 0 ) {  // not end of list
         tempPtr = currentPtr;
         cout << "Deleting " << tempPtr->data << endl;
         currentPtr = currentPtr->nextPtr; // move ptr
         tempPtr->nextPtr=0;
         delete tempPtr; // delete last node
      }
    cout << "All nodes destroyed\n\n";
   }

}
```

insertAtFront

```
// Insert a node at the front of the list
template<class T>
void List<T>::insertAtFront( const T &value )
{
    // create a new node with the value in it.
   Node<T> *newPtr = new Node<T>(value);
   assert(newPtr != 0);

   if ( head == 0 )  // if List is empty
       head = newPtr; // point to new node
   else {          // if List is not empty
       newPtr->nextPtr = head; // point to list
       head = newPtr; // move up firstPtr
   }

}
```

removeAtFront

```
// Delete a node from the front of the list
template<class T>
void List<T>::removeAtFront()
{
   if ( head == 0 )             // List is empty
       cout << "Error: list is empty!\n";
   else {
      Node<T> *tempPtr = head;
      cout <<"Deleting " << tempPtr->data << endl;
      head = head->nextPtr; // move to next node
      delete tempPtr;
       }

}
```

print

```
// Display the contents of the List
template<class T>
void List<T>::print() const
{
   if ( head == 0 ) {
      cout <<"The list is empty\n\n";
   }
   else   {
     Node<T> *currentPtr = head;
     cout <<"The list's Contents are : ";
     while ( currentPtr != 0 ) { // not end of list
         cout <<currentPtr->data << " -> ";
         currentPtr = currentPtr->nextPtr;
     }
     cout <<"\n";
   }
}
```

getSize

```
template<class T>
unsigned int List<T>::getSize() const
{
     unsigned int count=0;
   if ( head == 0 ) {
      return 0;
   }
   else   {
     Node<T> *currentPtr = head;
      while ( currentPtr != 0 ) { // not end of list
         count++;
         currentPtr = currentPtr->nextPtr;
     }
   return count;
   }
}
```

CopyConstructor

```
// copy constructor
template<class T>
List<T>::List(const List<T> &rhs) {
   head = 0;
   Node<T> *ptr = rhs.head;
   Node<T> *newPtr, *lastPtr;
   while (ptr != 0) {
       // create a new node with the value in it.
       newPtr = new Node<T>(ptr->data);
       assert(newPtr != 0);
       if(head == 0)
         head = newPtr; // first node
       else
          lastPtr->nextPtr = newPtr; // set up last link
       lastPtr = newPtr; // save last ptr
       ptr = ptr->nextPtr; // move up ptr
  }
}
```

Assignment Operator

```
// assignment operator
template<class T>
List<T> & List<T>::operator =(const List<T> &rhs) {
   if (this != &rhs) {
       deleteList();
       head= 0;
       Node<T> *ptr = rhs.head;
       Node<T> *newPtr, *lastPtr;
       while (ptr != 0) {
         // create a new node with the value in it.
         newPtr = new Node<T>(ptr->data);
         assert(newPtr != 0);
         if(head == 0)
           head = newPtr; // first node
         else
            lastPtr->nextPtr = newPtr; // set up last link
         lastPtr = newPtr; // save last ptr
         ptr = ptr->nextPtr; // move up ptr
      }
  }
}
```

Driver Program

```
int main()
{
  List<int> L1;
  cout << "The size of the list is: " << L1.getSize() <<
endl;
  L1.insertAtFront(123);
  L1.insertAtFront(456);
  L1.insertAtFront(789);
  cout << "The size of the list now is: " << L1.getSize()
<< endl;
  L1.print();
  L1.removeAtFront();
  cout << "The size of the list now is: " << L1.getSize()
<< endl;
  List<int> L2;
  L2=L1;
  L2.insertAtFront(134);
  L2.print();
  cout << "The size of the list now is: " << L2.getSize()
<< endl;
  system("pause");
  return 0;
}
```