Linked Allocation

Arrays, and structures housed in arrays, are *static structures*. A *dynamic data structure* is one that is allowed to expand and contract dynamically in its place in memory while the program is running. A dynamic data structure is a collection of elements, called **nodes**, explicitly ordered by means of **links** (pointers). A node may be viewed as a record structure.

Disadvantages of static structures:

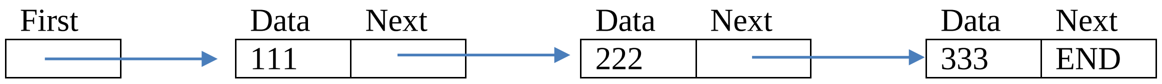
- full storage remains allocated during the entire run even though only a small amount may actually be used
- there is the ever-present possibility of overflow
- it is difficult or impossible for two or more structures to share the same allocation so that what is not used by one can be used by the other
- insertions to or deletions from the middle of an ordered array means moving many elements to either make room for the insertion or close the gap caused by the deletion

Static storage uses *implicit ordering*.
Dynamic storage uses *explicit ordering*.

Implicit ordering – items ordered by their sequential ordering in the array
Explicit ordering – each item contains within itself the address (pointer to location) of the next item.

Linked Lists

Each item in a linked list is called a node, and this node is an object contains at least two data members: the “data” and “next”, i.e., a link to the next node in the list. An external pointer contains the address of the first node in the list. The Next link of the last node in the list contains a null value to indicate the end of the list.



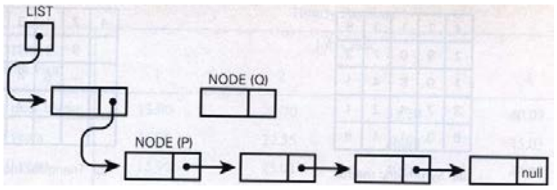
An empty list contains no nodes. the value of the external pointer is null.

```
class Node {
public:
      Node (int);
      void setData (int);
      int getData () const;
      void setNextPTR (const Node *);
      const Node * getNextPTR() const;
private:
      int Data;
      Node *NextPTR;    //points to “itself”
};
```

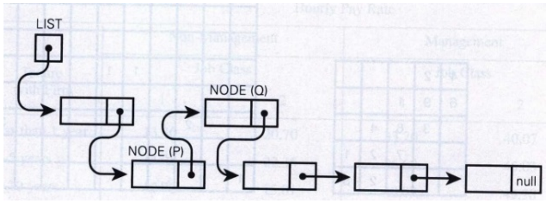
A *self-referential class* is one that contains a pointer member that points to a class object of the same class type.

When inserting into a linked list the item may be inserted directly into its proper place in the list without moving all the elements to make room for it.

Before insertion:



After insertion:



Disadvantages of linked allocation:

- extra space is required for the links
- possibly longer search time

How do we obtain a new node from the pool (guess what? these may be maintained on a stack) of available storage locations? The **new** keyword.

Example:

```
//linkedlist.cpp
//first attempt at processing a linked list
//list is in no particular order; items are added at the front
#include <iostream>
//using namespace std;

class node{
public:

   void setData(int);
   int getData();
   void setNextPtr ();
   node* getNextPtr();
public: // This should be private; but this is a primitive first attept
    int data;
   node* nextPtr;
};

int main(){
    node* firstPtr = NULL;        // creates a new empty list
    int val;
   node* p;
   cout << "To end this program enter a negative number.\n\n";


   cout << "Enter a positive integer:  "; cin >> val;
   while (val >= 0) {            // read and add values to the list
    p = new node;
    p->data = val;
      p->nextPtr = firstPtr;
      firstPtr = p;
      cout << "Enter a positive integer:  "; cin >> val;
   }//end while
   cout << "\nNow I'll print out your list from first item to last.\n";
   p = firstPtr;
   while (p != 0){
    cout << p->data << endl;
      p = p->nextPtr;
   }
   char c; cin>>c; //holds console window
   return 0;
}
```

```
To end this program enter a negative number.

Enter a positive integer:  10
Enter a positive integer:  20
Enter a positive integer:  30
Enter a positive integer:  40
Enter a positive integer:  55
Enter a positive integer:  -1

Now I'll print out your list from first item to last.
55
40
30
20
10
```

Program 2:

```
//linkedlist1.cpp
//first attempt at processing a linked list
//list is in no particular order; items are added at the front
#include <iostream>
//using namespace std;

class node{
public:

   void setData(int);
   int getData();
   void setNextPtr ();
   node* getNextPtr();
public: // This should be private; but this is a primitive first attempt
    int data;
   node* nextPtr;
};

int main(){
    node* firstPtr = NULL;        // creates a new empty list
    int val;
   node* p;

   //read and add values to the list
   cout << "To end this program enter a negative number.\n\n";
   cout << "Enter a positive integer:  "; cin >> val;
   while (val >= 0) {
    p = new node;
    p->data = val;
      p->nextPtr = firstPtr;
      firstPtr = p;
      cout << "Enter a positive integer:  "; cin >> val;
   }//end while

   //print all elements in the list first to last
   cout << "\nNow I'll print out your list from first item to last.\n";
   p = firstPtr;
   while (p != 0){
    cout << p->data << endl;
      p = p->nextPtr;
   }
   //sequential search of linked list
   char c;
   cout << "Do you wish to search for a target value? (y/n)" ; cin >> c;
   while (c != 'n'){
    cout << "\nEnter target:  "; cin >> val;
    p = firstPtr;
    while (p != NULL) {
                           if (p->data == val) break;
                           p = p->nextPtr;
    }
                           if (p == NULL) cout << "target not found" << endl;
                           else cout << "target found at location " << p <<
endl;

      cout << "Do you wish to search for a target value? (y/n)" ; cin >> c;
   }

   //end of program
   {char c; cin>>c;} //holds console window
   return 0;
}
```

Program:

```
//linkedlist2.cpp
//working with a linked list

#include <iostream>

class node{
public:
    //constructor, destructor
   void setData(int);   // get and set functions
   int getData();
   void setNextPtr ();
   node* getNextPtr();
public: // This should be private; but this is a primitive first attept
    int data;
   node* nextPtr;
};

int main(){
    node* firstPtr = NULL;        // creates a new empty list
    node* lastPtr = NULL;
   int val;
   node * p, * q, * pnew;

   // read and add new values to the list
   cout << "To end program enter a negative number. \n";
   cout << "Enter a positive integer:  "; cin >> val;
   while (val >= 0) {
    pnew = new node;
      pnew->data = val;

      if (firstPtr == NULL){     //list is empty
                           firstPtr = lastPtr = pnew;
         pnew->nextPtr = NULL;
         }
                           else {
                                  //search for position of new value in list
```

```
                              p = firstPtr;
                              while (p != NULL){
                              if (p->data > val) break;
                                     q = p;
                              p = p->nextPtr;
                                     }
                              //insert in proper position
                              if (p == firstPtr){        //new first
                                     pnew->nextPtr = firstPtr;
                              firstPtr = pnew;
                                     }
                              else if (p == NULL){       //new last
                                     pnew->nextPtr = NULL;
                              lastPtr->nextPtr = pnew;
                              lastPtr = pnew;
                              }
                              else {                             //insert before p
and after q
                                     pnew->nextPtr = p;
                              q->nextPtr = pnew;
                              }
               }
               cout << "Enter a positive integer:  "; cin >> val;
                              }

   //print all elements in the list first to last
   cout << "\nNow I'll print out your list from first item to last.\n";
   p = firstPtr;
   while (p != 0){
    cout << p->data << endl;
       p = p->nextPtr;
    } //

   //end of program
   char c; cin>>c; //holds console window
   return 0;
}   //
```

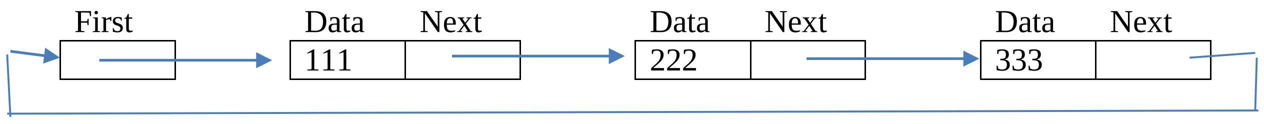
Program:

```
//linkedlist3.cpp
//first attempt at processing a linked list
//list is in no particular order; items are added at the front
#include <iostream>
//using namespace std;

class node{
public:
    //constructor, destructor funs
   void setData(int);
   int getData();
   void setNextPtr ();
   node* getNextPtr();
public: // This should be private; but this is a primitive first attempt
    int data;
   node* nextPtr;
};

void printup (node*);
void printdown (node*);

int main(){
   node* firstPtr = NULL;        // creates a new empty list
   int val;
   node* p;

   //read and add values to the list
   cout << "To end this program enter a negative number.\n\n";
   cout << "Enter a positive integer:  "; cin >> val;
   while (val >= 0) {
   p = new node;
   p->data = val;
      p->nextPtr = firstPtr;
      firstPtr = p;
      cout << "Enter a positive integer:  "; cin >> val;
   }//end while

   //print all elements in the list first to last
   cout << "\nNow I'll print out your list from first item to last.\n";
   printup(firstPtr);

   //print all elements in the list last to first
   cout << "\nNow I'll print out your list from last item to first.\n";
   printdown(firstPtr);


   //end of program
   {char c; cin>>c;} //holds console window
   return 0;
}


void printup (node* p){
   if (p == NULL) return;
   else {
                           cout << p->data << endl;
                           printup (p->nextPtr);
      }
}

void printdown (node* p){
   if (p == NULL) return;
   else {
                           printdown (p->nextPtr);
         cout << p->data << endl;
      }
}
```

Special case linked lists

Circular linked list.

In this structure, the LAST element in the list points back to the FIRST. We can enter the list (say, to search) anywhere in the "ring."

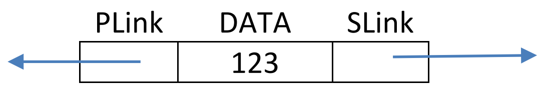


It may be easier to represent a queue as a circular list than as a linear list. We don't need both FRONT and REAR pointers, only one external pointer, pointing to the rear. Then the following node is the front.

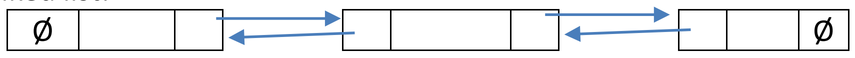
Doubly linked list.

A doubly-linked list has both forward and backward pointers, to successor and predecessor nodes. This allows us to (a) print (search, etc.) a list in reverse order without having to re-sort (b) backtrack.

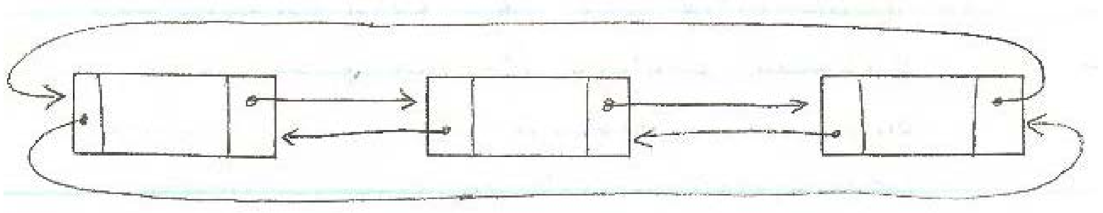
Node in a double-linked list:



A doubly-linked list:



And, then, a circular doubly-linked list would look like this:

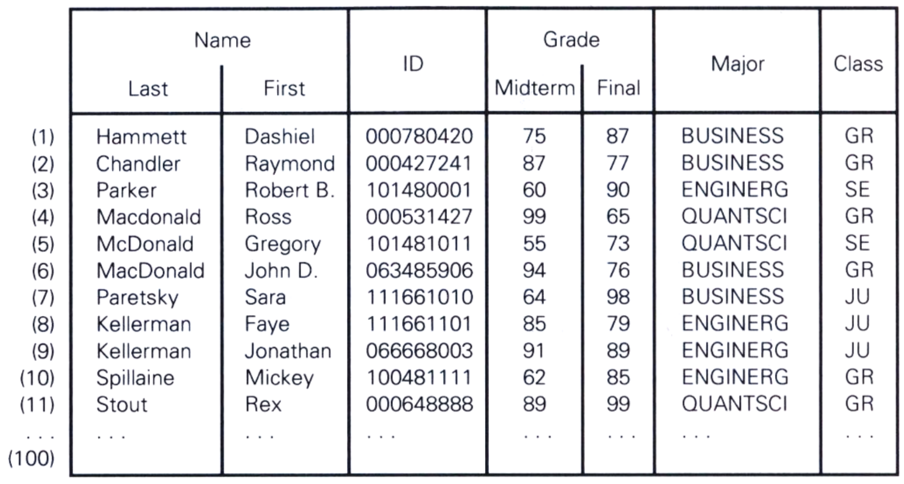


Multilinked Lists

Each element in the list has two or more pointers useful for (a) ordering the list on several different keys (b) creatingand maintaining sublists. For example, a contact list may be ordered on name, ID, relationship, physical address, zip code, area code, etc.

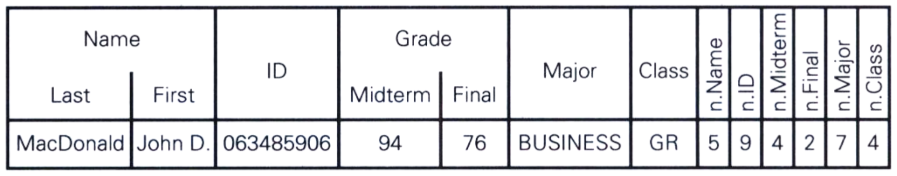
Without the multiple links we might use separate physically ordered sequential lists, introducing redundancy and the potential loss of data integrity.

Example of a multilinked list:



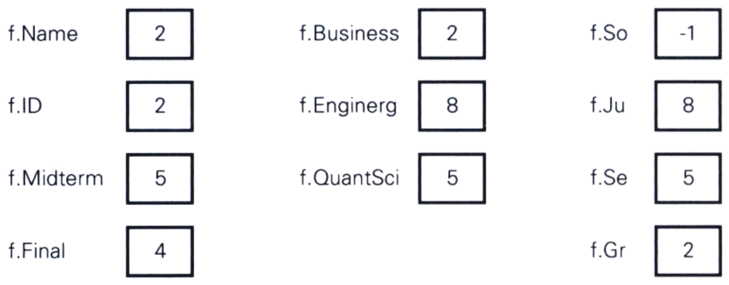
	Name		ID	Grade		Major	Class
	Last	First		Midterm	Final		
(1)	Hammett	Dashiel	000780420	75	87	BUSINESS	GR
(2)	Chandler	Raymond	000427241	87	77	BUSINESS	GR
(3)	Parker	Robert B.	101480001	60	90	ENGINERG	SE
(4)	Macdonald	Ross	000531427	99	65	QUANTSCI	GR
(5)	McDonald	Gregory	101481011	55	73	QUANTSCI	SE
(6)	MacDonald	John D.	063485906	94	76	BUSINESS	GR
(7)	Paretsky	Sara	111661010	64	98	BUSINESS	JU
(8)	Kellerman	Faye	111661101	85	79	ENGINERG	JU
(9)	Kellerman	Jonathan	066668003	91	89	ENGINERG	JU
(10)	Spillaine	Mickey	100481111	62	85	ENGINERG	GR
(11)	Stout	Rex	000648888	89	99	QUANTSCI	GR
. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .
(100)							

(a) The data set



Name		ID	Grade		Major	Class	n.Name	n.ID	n.Midterm	n.Final	n.Major	n.Class
Last	First		Midterm	Final								
MacDonald	John D.	063485906	94	76	BUSINESS	GR	5	9	4	2	7	4

(b) A single node in the list (includes "next" links)



f.Name	2	f.Business	2	f.So	-1
f.ID	2	f.Enginerg	8	f.Ju	8
f.Midterm	5	f.QuantSci	5	f.Se	5
f.Final	4			f.Gr	2

(c) External pointers to first in each list