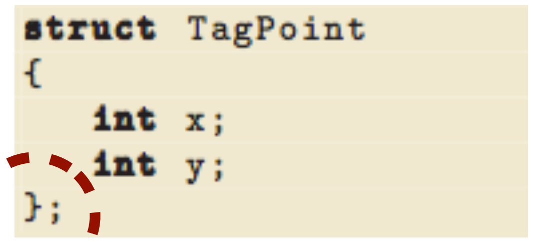
Structures, linked lists, queues and stacks

Structures

What is a structure?

- A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling.

Don't forget the semi-colon!



```
struct TagPoint
{
    int x;
    int y;
};
```

- **struct** is a keyword introducing the structure declaration, which is the list in braces { }.

- **TagPoint** is an optional structure tag. It names this kind of structure and may be used as shorthand for the part of the declaration in braces. The labels **x** and **y** are called members of the structure. They are "within the scope" of this structure.

Declaring a structure

- Note that a structure declaration followed by no variables does not allocate any space; it just defines the template for later use.

```
struct TagPoint
{
    int x;
    int y;
};
struct TagPoint point;
struct TagPoint maxPoint = {320, 200};
```

- This code presented will use and allocate the space for the structure definition previously stated. The second variable maxPoint will declare a variable and initialize all members.

```
maxPoint.x == 320;
maxPoint.y == 200;
```

- The . operator connects the structure variable name and the member name. So, maxPoint.x == 320 will return TRUE.

A more complex example of structures

```
struct TagPoint
{
    int x;
    int y;
};
```

```
struct TagRectangle
{
   struct TagPoint upperLeftCorner;
   struct TagPoint lowerRightCorner;
};

struct TagRectangle polygon;

polygon.upperLeftCorner.x = 320;
```

```
struct TagPoint add (struct TagPoint p1, struct TagPoint p2)
{
   struct TagPoint temp;

   temp = p1;

   temp.x += p2.x;
   temp.y += p2.y;

   return (temp);
}
```

- This code works fine, but consider all of the memory getting copied to/from the stack; especially in cases when the structure has several members.
- A more efficient alternative is to pass the address of the structure variables, and then use pointers within the function. Pointers to structures are just like any other pointer variables.

```
#include <iostream>
using namespace std;

struct TagPoint { int x; int y; };

struct TagPoint bad_add (struct TagPoint p1, struct TagPoint p2)
{
     struct TagPoint temp;
     temp = p1;
     temp.x = temp.x + p2.x;
     temp.y = temp.y + p2.y;

     return(temp);
}

struct TagPoint good_add (struct TagPoint *p1, struct TagPoint *p2)
{
     struct TagPoint temp;
     temp = *p1;
     temp.x = temp.x + (*p2).x;
     temp.y = temp.y + (*p2).y;

     return(temp);
}

int main()
{
     TagPoint point1, point2, sumPoints;

     point1.x=10; point1.y=20;
     point2.x=10; point2.y=20;

     sumPoints = bad_add (point1, point2);
     cout<<sumPoints.x<<" "<<sumPoints.y<<endl;

     sumPoints = good_add (&point1, &point2);
     cout<<sumPoints.x<<" "<<sumPoints.y<<endl;
}
```

Note: this is C++ code, and it will not work on the Arduino (especially the cout, namespace and iostream library)

Pointers to structures

```
struct TagPoint
{
    int x;
    int y;
};
```

```
struct TagRectangle
{
    struct TagPoint upperLeftCorner;
    struct TagPoint lowerRightCorner;
};

struct TagRectangle polygon;

polygon.upperLeftCorner.x = 320;
```

```
struct TagPoint add (struct TagPoint p1, struct TagPoint p2)
{
    struct TagPoint temp;

    temp = p1;

    temp.x += p2.x;
    temp.y += p2.y;

    return (temp);
}
```

```
struct TagPoint *pp;
struct TagPoint p3;

pp = &p3;

/* The following are equivalent. */
(*pp).x == pp->x;
```

Pointers to structures are so common that a special operator **->** is used to dereference the member of a structure via a pointer.

```
struct TagRectangle r;
struct TagRectangle *rp;

rp = &r;
r.upperLeftCorner.x = 5;

/* The following are all equivalent. */
rp->upperLeftCorner.x == 5;
(r.upperLeftCorner).x == 5;
(rp->upperLeftCorner).x == 5;
```

Initializing structures

```c
#include <stdio.h>

struct date
{
    int month;
    int day;
    int year;
};

int main()
{
  struct date cdate;

  cdate.month=1;
  cdate.day=21;
  cdate.year=2012;

  printf("%i%i%i\n",cdate.month,cdate.day,cdate.year);

  return 0;
}
```

```c
#include <stdio.h>

struct date
{
    int month;
    int day;
    int year;
} cdate;

int main()
{
  cdate.month=1;
  cdate.day=21;
  cdate.year=2012;

  return 0;
}
```

Besides the missing printf, these two codes are functionally identical ...

Initializing structures with assignments

```c
#include <stdio.h>

struct date
{
    int month;
    int day;
    int year;
} cdate={1,21,2012};

int main()
{
  printf("%i%i%i\n",cdate.month,cdate.day,cdate.year);

  return 0;
}
```

```c
#include <stdio.h>

struct date
{
    int month;
    int day;
    int year;
} cdate;

int main()
{
  cdate.month=1;
  cdate.day=21;
  cdate.year=2012;

  return 0;
}
```

Besides the missing printf, these two codes are functionally identical ...

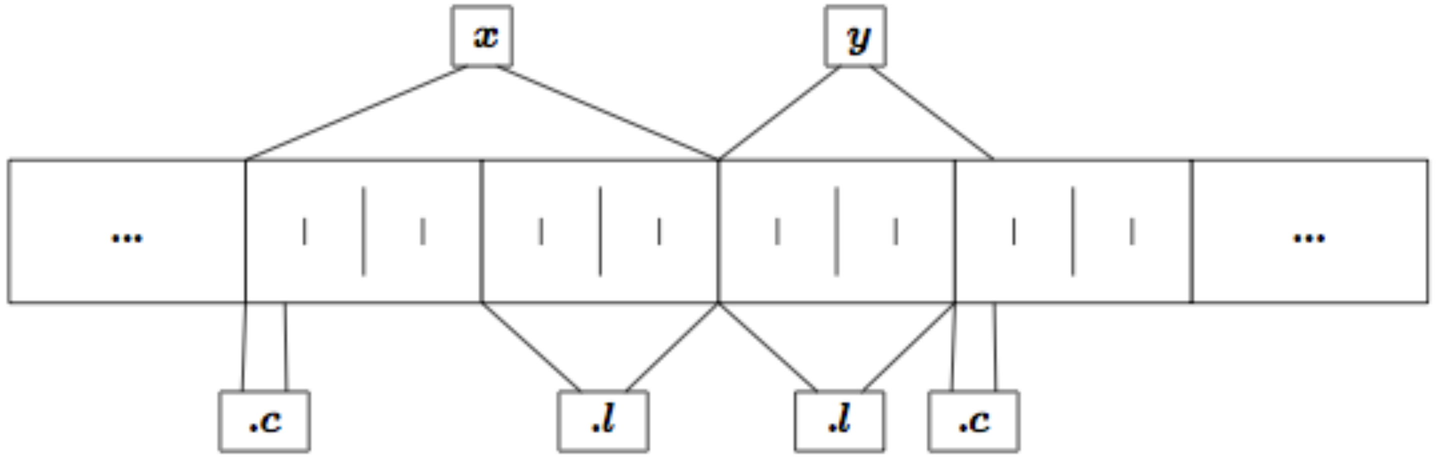
Initializing multiple structures

```
#include <stdio.h>

struct date
{
    int month;
    int day;
    int year;
} cdate={1,21,2012}, newdate2, newdate3={1,15,1976};

int main()
{
  printf("%i%i%i\n",cdate.month,cdate.day,cdate.year);

  newdate2.month=1;
  newdate2.day=21;
  newdate2.year=2012;

  printf("%i%i%i\n",newdate3.month,newdate3.day,newdate3.year);

  return 0;
}
```

Memory allocation of structs

```
#include <stdio.h>

struct x
{
    char c;
    long l;
};

struct y
{
    long l;
    char c;
};

int main()
{

  return 0;
}
```

- **x** requires 8 bytes
- **y** only uses 5 bytes

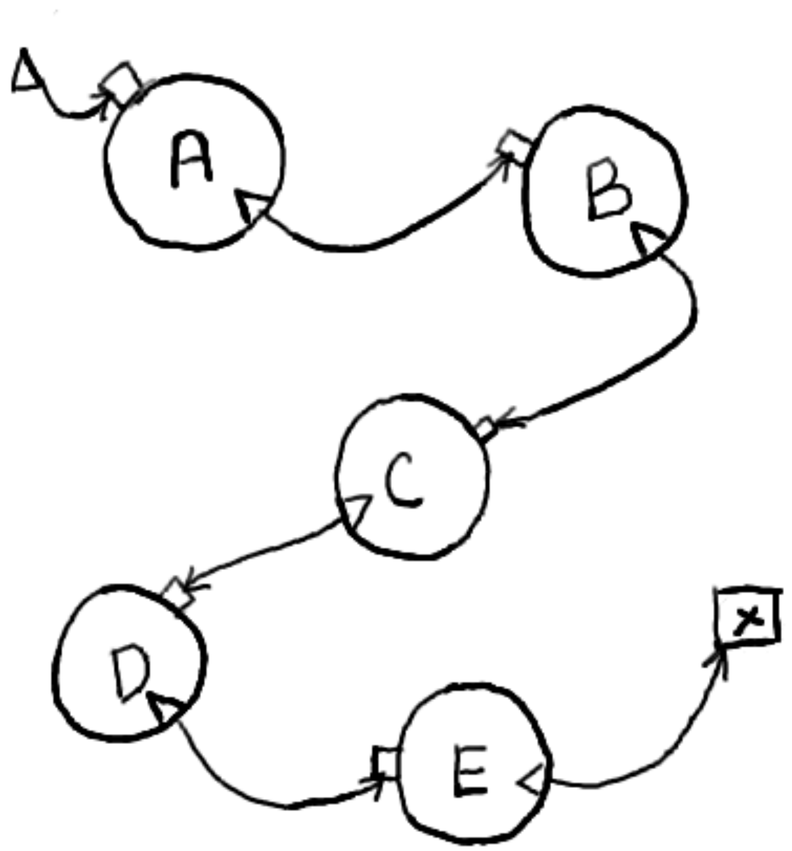


Linked lists

reference: http://codingfreak.blogspot.com/2009/08/implementation-of-singly-linked-list-in.html

What is a linked list?

- A Linked list is a chain of structs called Nodes.
- Each node has at least two members, one of which points to the next Node in the list and the other holds the data.
- These are defined as Single Linked Lists because they can only point to the next Node in the list but not to the previous.



Linked List

Basic element of a linked list

```
struct Node
{
  int Data;
  struct Node *Next;
}*Head;
```

*Head will become a global pointer.

- This structure defines a Node.
- Variable **Data** holds the data in the Node while pointer of type struct Node **Next** holds the address to the next Node in the list.
- I create an initialed Node structure named ***Head** which is a pointer of type struct Node.
- This acts as the Head to the list.
- In the main program, we initially we set 'Head' as NULL which means list is empty.

Function that inserts data at the start of the linked list

```
struct Node
{
  int Data;
  struct Node *Next;
}*Head;
```

```
void addBeg(int num)
{
   struct Node *temp;

   temp=(struct Node *)malloc(sizeof(struct Node));
   (*temp).Data = num;

   if (Head == NULL)
   {
      //List is Empty
      Head=temp;
      (*Head).Next=NULL;
   }
   else
   {
      (*temp).Next=Head;
      Head=temp;
   }
}
```

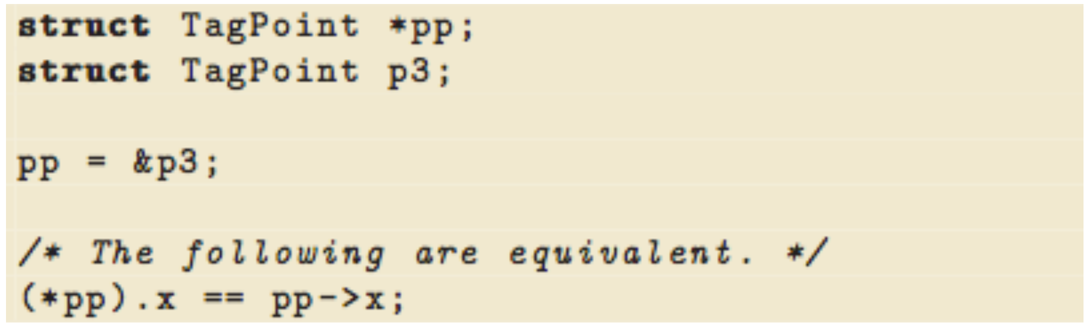
- Inside the function we dynamically allocate memory
- We create 'temp' node and save the Data part.
- If Head is NULL it means list is empty. So we set temp node as the Head of the list and set the Next as NULL.
- Else we set the Next in temp node as Head and reassign Head with temp.

Function that inserts data at the start of the linked list

```
struct Node
{
   int Data;
   struct Node *Next;
}*Head;
```

```
// Adding a Node at the Beginning of the List
void addBeg(int num)
{
   struct Node *temp;

   temp=(struct Node *)malloc(sizeof(struct Node));
   temp->Data = num;

   if (Head == NULL)
   {
      //List is Empty
      Head=temp;
      Head->Next=NULL;
   }
   else
   {
      temp->Next=Head;
      Head=temp;
   }
}
```

- Same code as before, but in a more readable format.
- Remember:

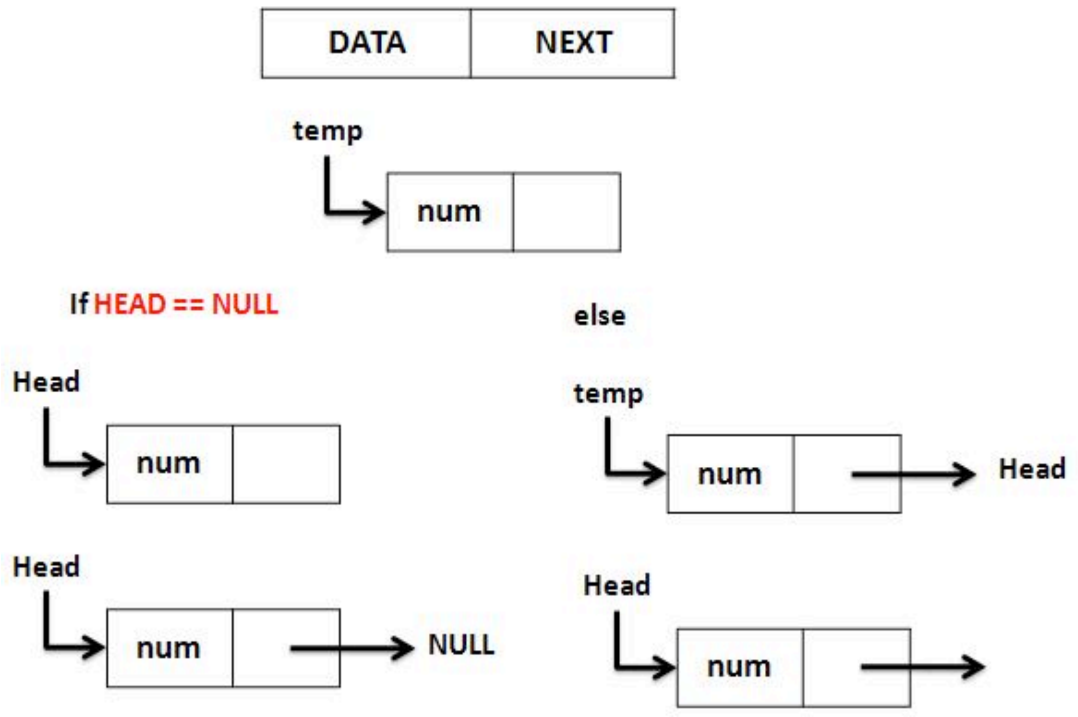


```
struct TagPoint *pp;
struct TagPoint p3;

pp = &p3;

/* The following are equivalent. */
(*pp).x == pp->x;
```

Function that inserts data at the start of the linked list

```
struct Node
{
  int Data;
  struct Node *Next;
}*Head;
```

```
// Adding a Node at the Beginning of the List
void addBeg(int num)
{
   struct Node *temp;

   temp=(struct Node *)malloc(sizeof(struct Node));
   temp->Data = num;

   if (Head == NULL)
   {
      //List is Empty
      Head=temp;
      Head->Next=NULL;
   }
   else
   {
      temp->Next=Head;
      Head=temp;
   }
}
```



WESTERN NEW ENGLAND UNIVERSITY WNE

Dynamic memory allocation

- If you want to dynamically allocate memory outside Arduino (for example in gcc or Microsoft visual studio you need the following two libraries:
  - #include <stdio.h>
  - #include <stdlib.h>

Display the entire linked list contents

```
void display()
{
  struct Node *cur_ptr;

  cur_ptr=Head;

  if(cur_ptr==NULL)
  {
     printf("\nList is Empty");
  }
  else
  {
      printf("\nElements in the List: ");
      //traverse the entire linked list
      while(cur_ptr!=NULL)
      {
          printf(" -> %d ",cur_ptr->Data);
          cur_ptr=cur_ptr->Next;
      }
      printf("\n");
  }
}
```

- Create a new pointer to the Node structure, and define its starting point to be the Head of the structure.
- If the Head pointer is NULL, then the list is empty
- Otherwise, keep cycling every single node.

Putting these two functions together (add elements and display them)

```
 struct Node
 {
   int Data;
   struct Node *Next;
 }*Head;
```

```
// Adding a Node at the Beginning of the List
void addBeg(int num)
{
 struct Node *temp;

 temp=(struct Node *)malloc(sizeof(struct Node));
 temp->Data = num;

 if (Head == NULL)
 {
      //List is Empty
      Head=temp;
      Head->Next=NULL;
 }
 else
 {
      temp->Next=Head;
      Head=temp;
 }
}
```

```
void display()
{
  struct Node *cur_ptr;
  cur_ptr=Head;

  if(cur_ptr==NULL)
  {
     printf("\nList is Empty");
  }
  else
  {
      printf("\nElements in the List: ");
      //traverse the entire linked list
      while(cur_ptr!=NULL)
      {
          printf(" -> %d ",cur_ptr->Data);
          cur_ptr=cur_ptr->Next;
      }
      printf("\n");
  }
}
```

```
int main()
{
 Head=NULL; //Set HEAD as NULL
 addBeg(10); addBeg(8); addBeg(3);
 display(); //3 -> 8 -> 10
}
```

Determining the length of our linked list

```
 struct Node
 {
   int Data;
   struct Node *Next;
 }*Head;
```

```
// Adding a Node at the Beginning of the List
void addBeg(int num)
{
 struct Node *temp;

 temp=(struct Node *)malloc(sizeof(struct Node));
 temp->Data = num;

 if (Head == NULL)
 {
      //List is Empty
      Head=temp;
      Head->Next=NULL;
 }
 else
 {
      temp->Next=Head;
      Head=temp;
 }
}
```

```
//counting the number of elements in the list
int length()
{
  struct Node *cur_ptr;
  int count=0;

  cur_ptr=Head;

  while(cur_ptr != NULL)
  {
     cur_ptr=cur_ptr->Next;
     count++;
  }
  return(count);
}
```

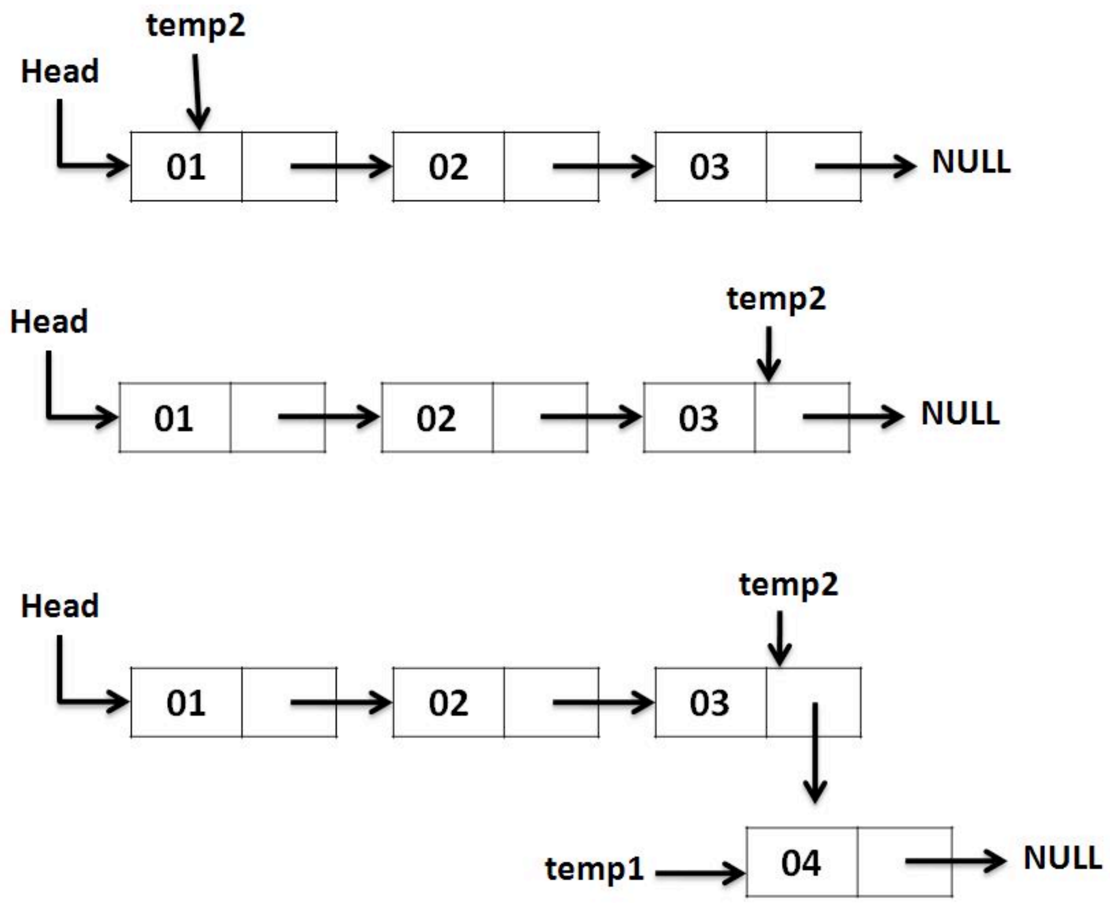
```
int main()
{
 Head=NULL; //Set HEAD as NULL
 addBeg(10); addBeg(8); addBeg(3);
 //the length of this list is 3
 printf("Number of nodes: %d",length());
}
```

Delete specific nodes to prevent memory leaks

```
void delNodeLoc(int loc)
{
  struct Node *prev_ptr, *cur_ptr;
  int i;
  cur_ptr=Head;

  if(loc > (length()) || loc <= 0)
  { printf("\nDeletion of Node at given location is not possible\n "); }
  else
  {
      if (loc == 1) // If the location is starting of the list
      {
          Head=cur_ptr->Next;
          free(cur_ptr);
          return 0;
      }
      else
      {
          for(i=1;i<loc;i++)
          {
              prev_ptr=cur_ptr;
              cur_ptr=cur_ptr->Next;
          }

          prev_ptr->Next=cur_ptr->Next;
          free(cur_ptr);
      }
  }
}
```

Adding a node at the end of the linked list

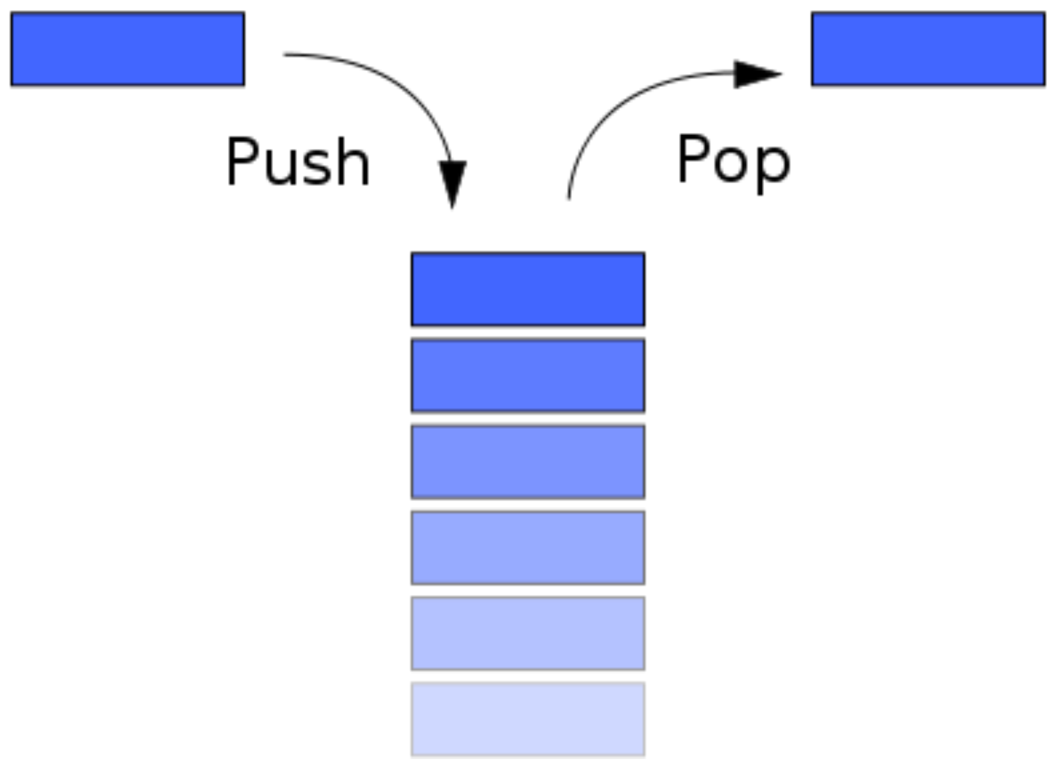
```
//Adding a Node at the end of the list
 void addEnd(int num)
 {
    struct Node *temp1, *temp2;

    temp1=(struct Node *)malloc(sizeof(struct Node));
    temp1->Data=num;

    // Copying the Head location into another node.
    temp2=Head;

    if(Head == NULL)
    {
       // If List is empty we create First Node.
       Head=temp1;
       Head->Next=NULL;
    }
    else
    {
       // Traverse down to end of the list.
       while(temp2->Next != NULL)
       temp2=temp2->Next;

       // Append at the end of the list.
       temp1->Next=NULL;
       temp2->Next=temp1;
    }
 }
```



Source code

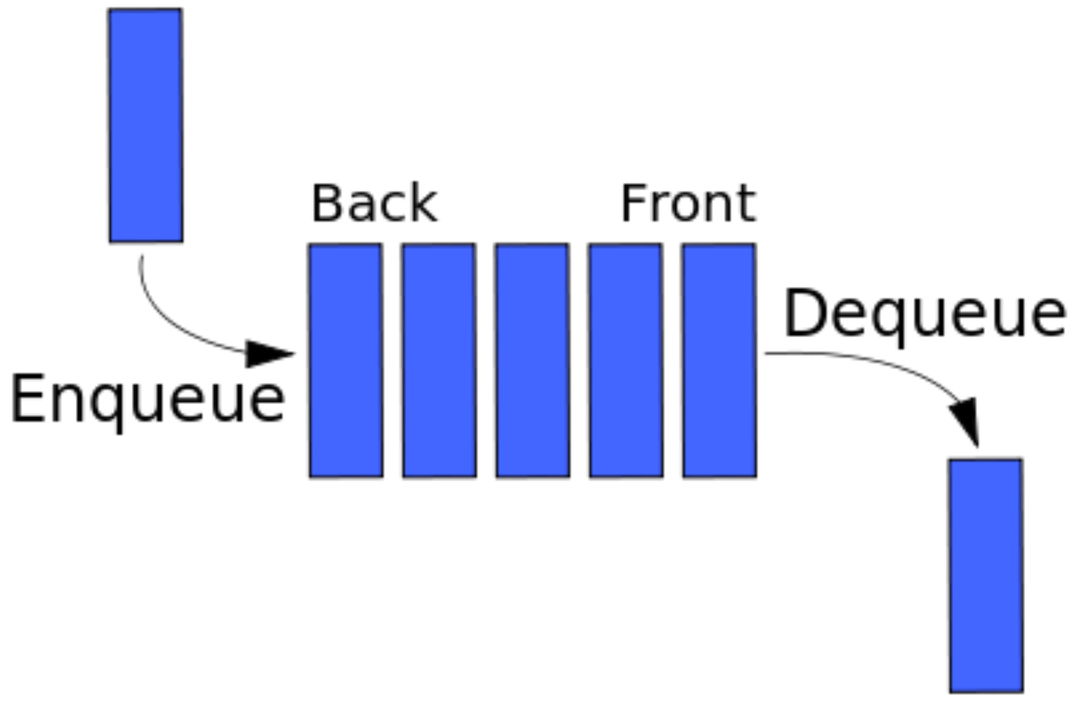
- Check out this excellent website: http://codingfreak.blogspot.com/2009/08/implementation-of-singly-linked-list-in.html
- It contains the full source of a linked list implementation

Stacks



- LIFO (Last In First Out)
- A stack can have any data type as an element, but is characterized by two fundamental operations; push and pop.
- The push operation adds a new item to the top of the stack, or initializes the stack if it is empty.

Queue



- FIFO (First In First Out)
- Is a kind of abstract data type in which the entities in the collection are kept in order and the only operations on the collection are the addition of entities to the rear position and removal of entities from the front position.
- In a FIFO data structure, the first element added to the queue will be the first one to be removed.