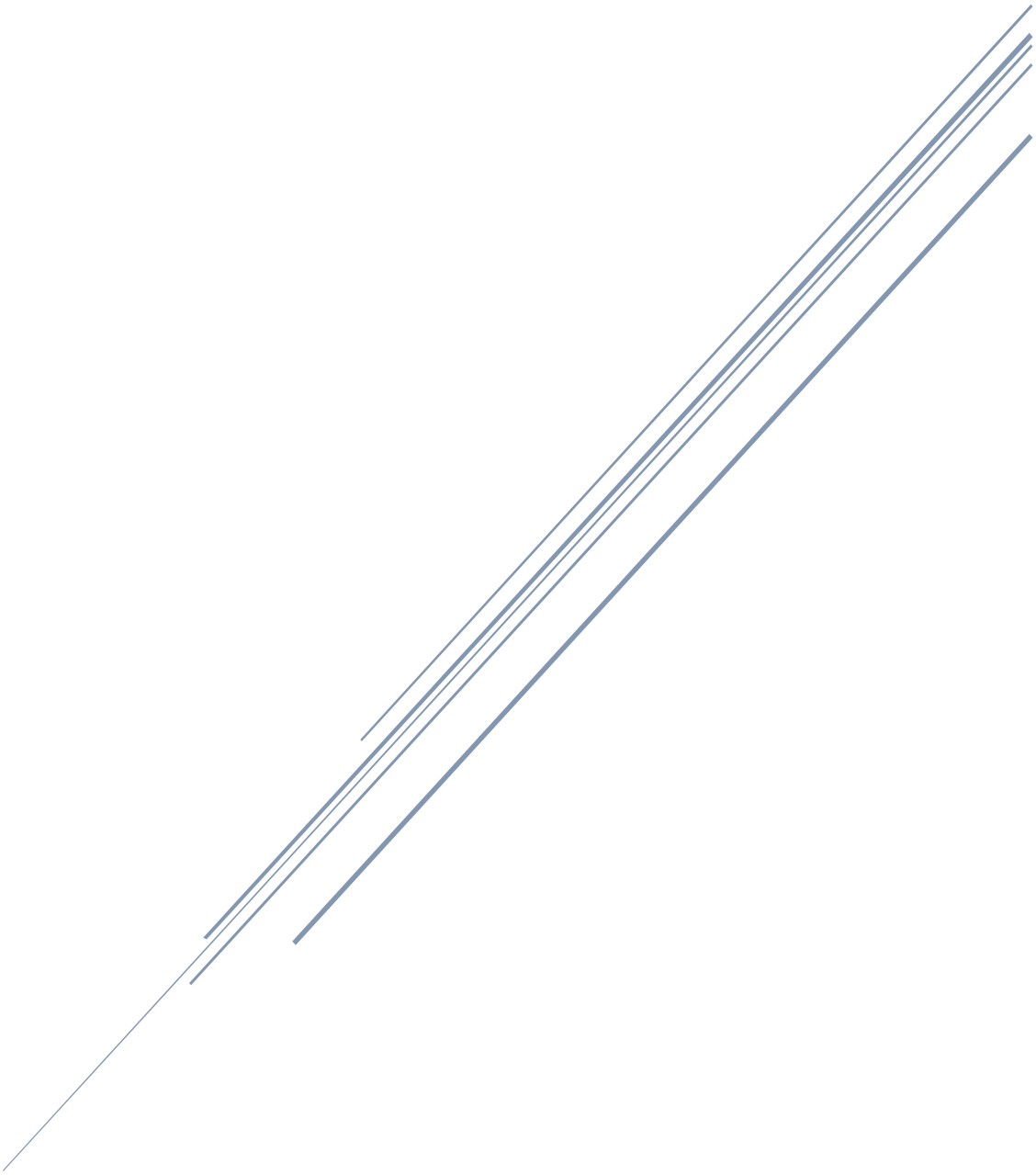
LISTS

Chapter 03



Prakash Hegade
Minor Degree Program – School of CSE, KLE Tech.

Lists

Introduction

The data structures we have seen so far have

- Fixed amount of storage (we have used arrays and static memory allocation)
- Have implicit ordering
    - If x is current location then x+1 gives the next location

What we shall see next is rather an alternate representation of the data. Not all the times the system can guarantee the implicit ordering. Let us say we requested for 10MB of memory and following is the memory status:

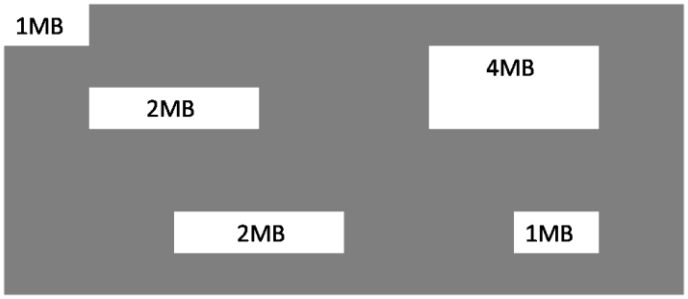


Figure: Memory Snapshot

As you see, we do have 10MB of memory. But they are not contagious. We don't want system to tell us that there is no memory available just because it is not contiguous. Instead we think of an alternative of how we can use this available space effectively.

Now we know that we don't need all this 10MB space at once all at once. We might need 1MB at a time. That puts us with the question, rather than asking for 10MB at once, can I ask 1MB at once for 10 times? Yes. We can. We shall do that. Let us understand the process with a little more of technical detail. Let us first ask for 1MB of data. Remember, 1 MB is huge for the textual data we are dealing with. It is only an example to understand. Otherwise all our operations are going to end up with few bytes.

The program makes a request for 1MB of data. Memory is allocated from the heap. Heap returns the starting address of this 1MB of data so that the program can make access to it.
Essentially, program should have a pointer to hold this address. Let us diagrammatically see how all this looks like.

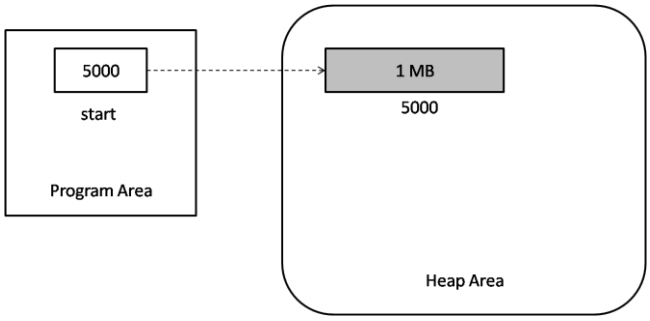


Fig: Memory Allocation of First Chunk

We needed 1MB of data and we have it in heap. The starting address of this 1MB is 5000 which we have captured through a pointer variable 'start'. We are all good and we go ahead with our program and suddenly at one fine time we run out of memory. We need 1MB more. We make a request again to heap and we get the memory allocated.

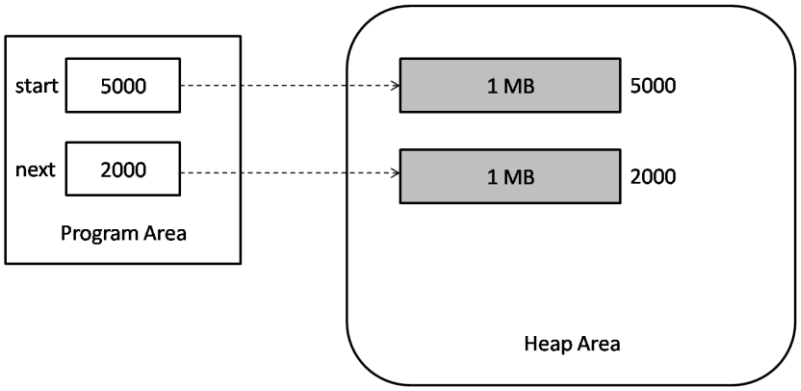


Figure: Memory Allocation of Second Chunk

Okay, this looks fine. We now have one more pointer called 'next' to hold the starting address of the next allocated chunk. But wait, there is a suggestion. Instead of growing up the number of pointers in the program for each allocated chuck, we can do the following way:

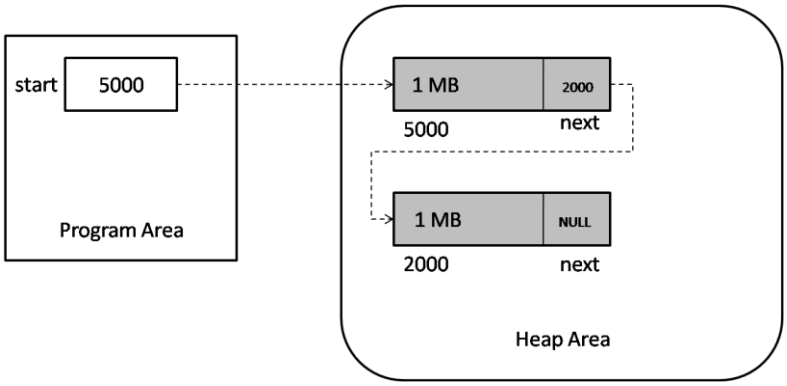


Figure: Alternate Design of Memory Allocation

This looks more promising. Our first 1MB data holds the address of where the next 1MB is and this process will continue ahead for the further chunks of memory allocated. Every memory allocated chunk has a variable which holds the address of where the next chuck is.

Each chuck will be officially named as 'Node' as looks like the following:

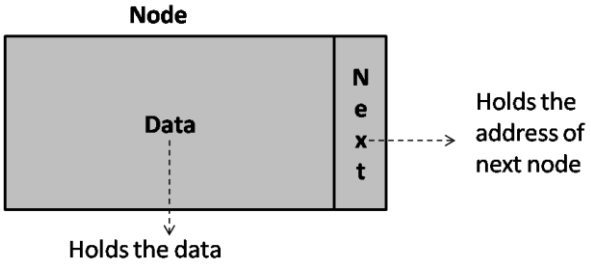


Figure: Structure of a Node

What we are going to have is a chain of nodes all linked together. Thus what we have is a Linked list. Get ready for this exciting and challenging journey. We are going to get a little dirty here. We are going to get messed up with pointers, more errors and more system crashes and yes, more knowledge. This is a turn where you will have a lot of interview questions and lot of applications in industry as well.

Understanding a Node

In this section we shall understand the technical details of a node and creating a code.

Here is how the **node** goes. We have a structure which houses the data and the pointer to next node. It's a self referential structure and it has a reference of its own type. What we mean is the next field holds the address of a node whose data type is same as itself AKA self referential. For now, let us assume we have integer data.

```
struct node
{
  int data;
  struct node *next;
};
typedef struct node NODE;
```

And the **memory allocation**:

```
NODE * newnode;
newnode = (NODE *) malloc( sizeof(NODE) );
```

Operations

We have a node with memory allocated. What kind of operations can we do?

- Add a new node
  - Where? Front? End? Or somewhere at required position?
- Delete a node
  - From where? Front? End? Or somewhere from required position?

Let us look at all the operations in detail one by one. Before we start with any, let us have few initial ground works done.

We have the structure:

```
struct node
{
  int data;
  struct node *next;
};
typedef struct node NODE;
```

Let us create a variable to hold the starting address of the list. Let us call it start and initialize it to NULL.

```
NODE *start = NULL;
```

Adding a New Node

Like already said a new node can be added at following positions

- Front
- End
- Required position

Adding a node at the front:

We have two cases. What if the list is empty? In case we assign the node address to the start pointer.

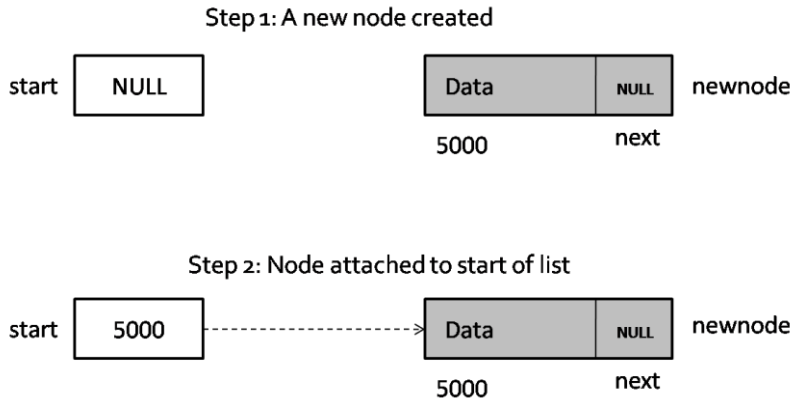


Figure: Adding a Node at Front – Case 1

If the list is not empty, we do the following:

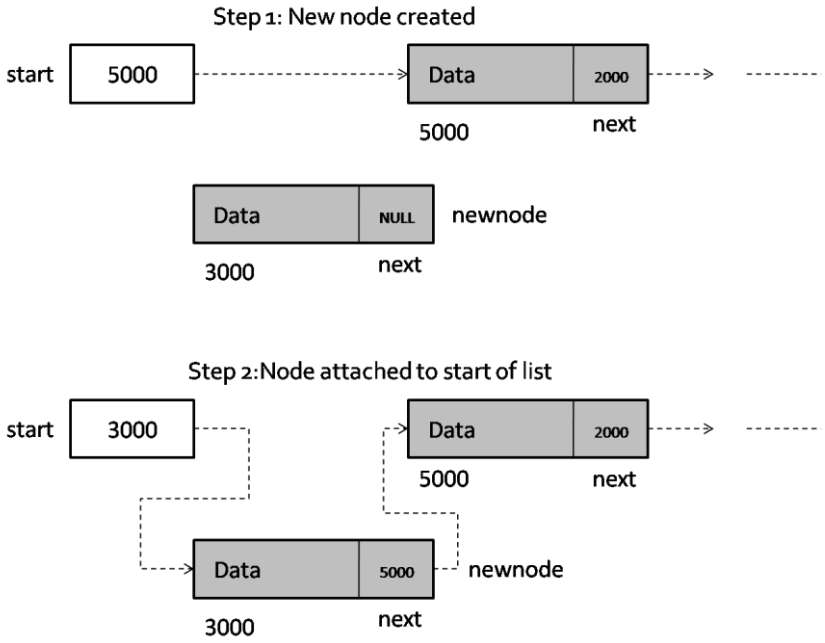


Figure: Adding a Node at Front – Case 2

Note how the pointer adjustments are done. We have two updates to make. One is the 'next' field of newly created node and other is the 'start' pointer.

Adding a node at the end:

We have two cases. What if the list is empty? In case we assign the node address to the start pointer. This is same as the case of front when the list is empty. As the list is empty, the only node we insert is the front and is the end.

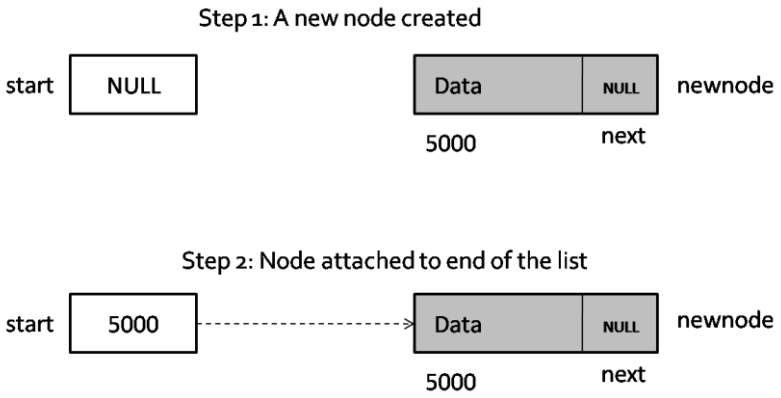


Figure: Adding a Node at End – Case 1

If the list is not empty, we do the following:

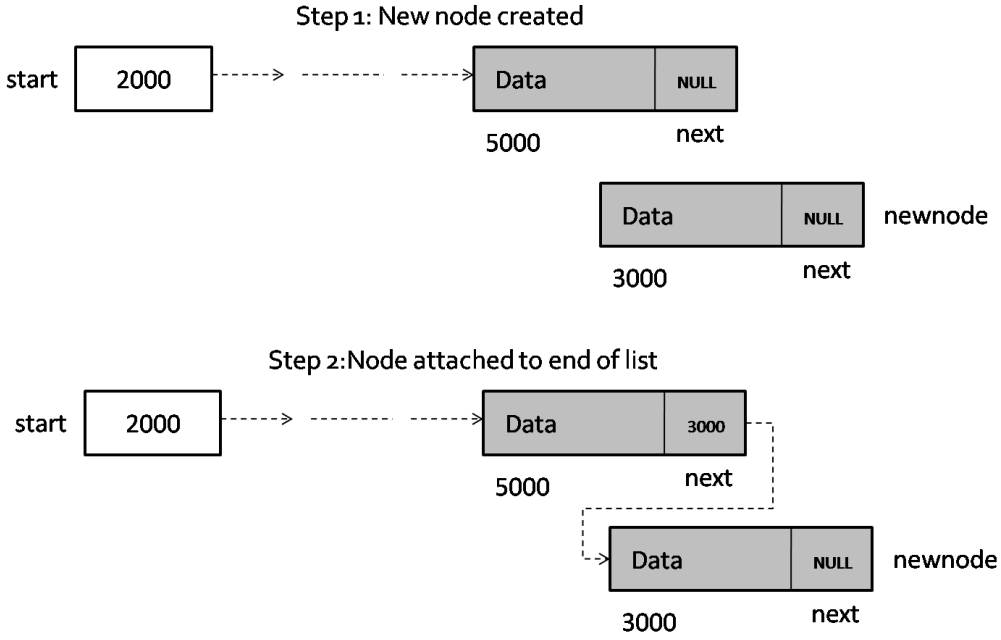


Figure: Adding a Node at End – Case 2

There are 'N' numbers of nodes and node with address 5000 is the last end. We attach the newly created node to end of this list. Look at how the 'next' field of node with address 5000 changes from 'NULL' to starting address '3000' which is of the newly created node.

Adding a node at the any position:
Any position can also be for first or last. We will skip those cases as we have already handled them.

We shall understand the case of adding the node at nth position. Look at the figure below. The pointer adjustments are quite intuitive.

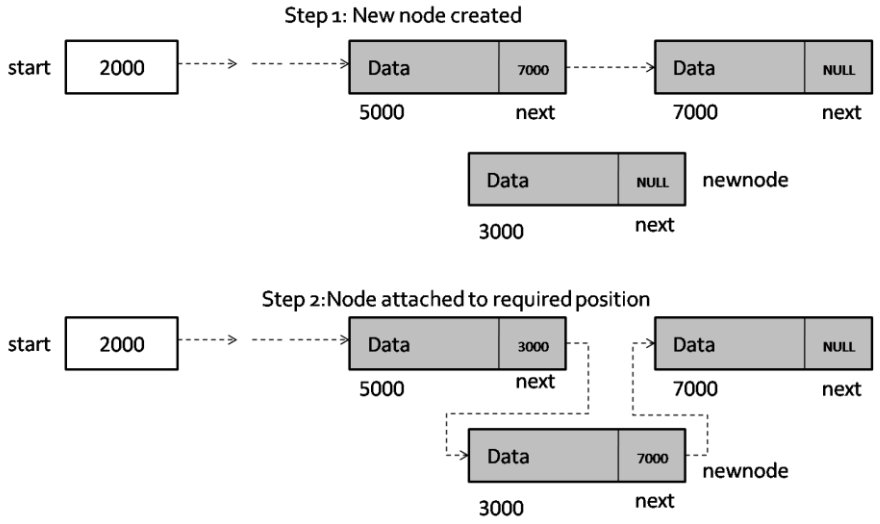


Figure: Adding a Node at position n

Deleting an Existing Node

Like already said an existing node can be deleted from following positions

- Front
- End
- Required position

Deleting a node from front:

We have two cases. What if it is the last node to be deleted and what if there are many other nodes. We shall handle both the cases.

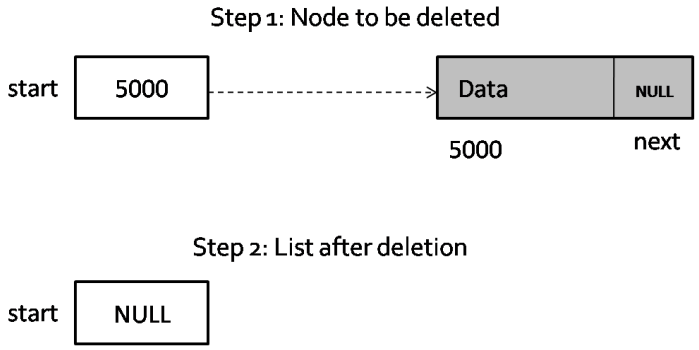


Figure: Deleting a Node from Front – Case 1

If the list has more than one node, we do the following pointer adjustments:

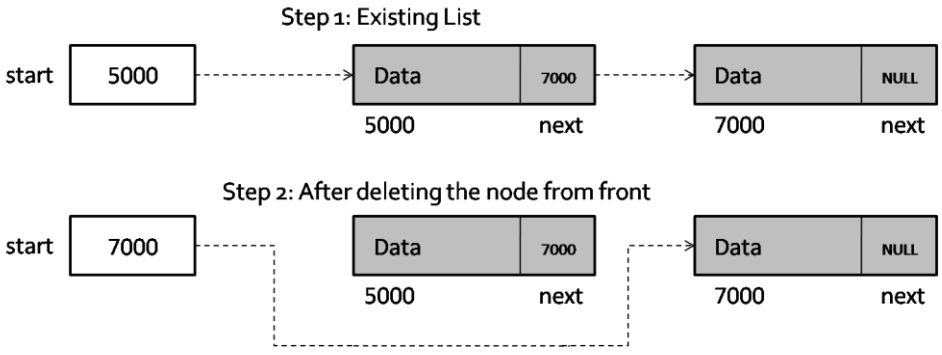


Figure: Deleting a Node from Front – Case 2

As you see, the start pointer is updated to the next node address. The next address can be obtained from the 'next' field of the first node.

Deleting a node from end:
We have two cases. What if it is the last node to be deleted and what if there are many other nodes. We shall handle both the cases.

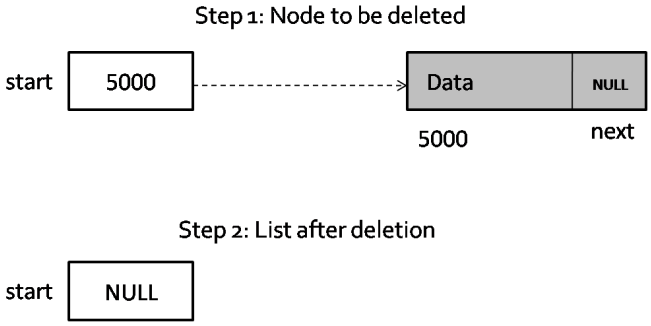


Figure: Deleting a Node from end – Case 1

As you notice this case is same as the case for delete from front. As there is only one node front and end refer to same node. Let us look at the other case.

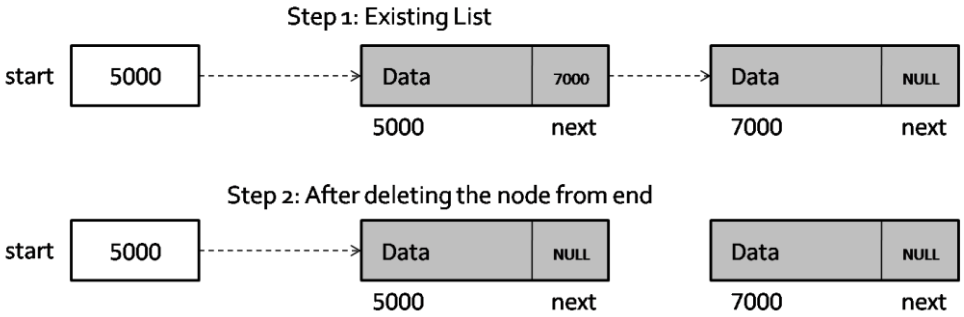


Figure: Deleting a Node from end – Case 2

Deleting a node from any position:
Any position can also be for first or last. We will skip those cases as we have already handled them. We shall look at the case of deleting a node from nth position. Look at the figure below. The pointer adjustments are quite intuitive.

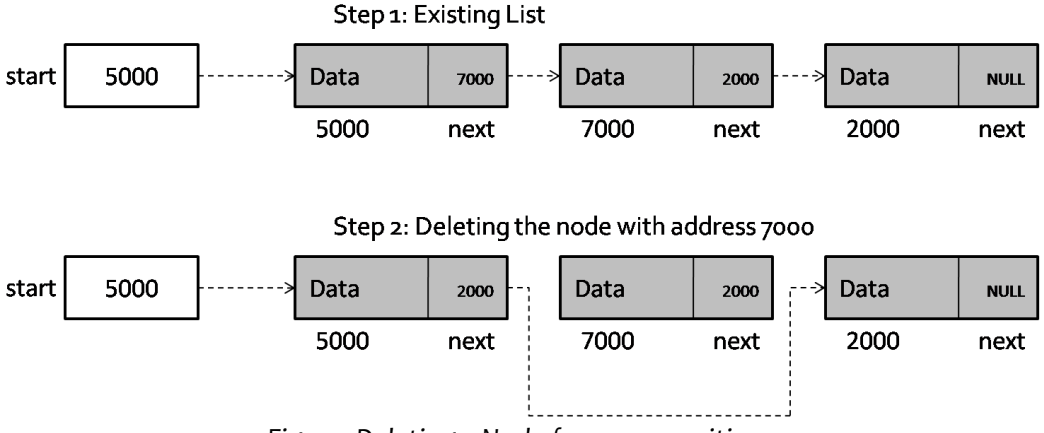


Figure: Deleting a Node from any position

The pointer adjustments are evitable from the diagram. Our next idea is to convert all these diagrams to the 'C' code.

Singly Linked List Implementation

```
#include <stdio.h>
#include <stdlib.h>
struct node
{
   int info;
   struct node *next;
};
typedef struct node NODE;

// Maintain the number of nodes in the list
int currnodes = 0;

NODE *  insert_at_start(NODE * start);
NODE *  insert_at_end(NODE * start);
NODE *  insert_at_position(NODE * start);
NODE *  delete_from_start(NODE * start);
NODE *  delete_from_end(NODE * start);
NODE *  delete_from_position(NODE * start);
NODE * getnode();
void getdata(NODE *);
void display_list(NODE *start);

int main() {
   NODE * start=NULL;
   int choice = 0;
   while(1)
   {
      printf("\n\n* * * * * *   Menu  * * * * * *\n");
      printf("1. Insert node at start\n");
      printf("2. Insert node at End\n");
      printf("3. Insert node at a Position\n");
      printf("4. Delete node from start\n");
      printf("5. Delete node from end\n");
      printf("6. Delete node from a Position\n");
      printf("7. Display List\n");
      printf("8. Exit\n");
      printf("* * * * * * ******** * * * * * *\n");
      printf("Enter your choice:\n");
      scanf("%d", &choice);

      switch (choice)
      {
         case 1: start = insert_at_start(start);
               break;
```

```
        case 2: start = insert_at_end(start);
                break;
        case 3: start = insert_at_position(start);
                break;
        case 4: start = delete_from_start(start);
                break;
        case 5: start = delete_from_end(start);
                break;
        case 6: start = delete_from_position(start);
                break;
        case 7: display_list(start);
                break;
        case 8: printf("Exiting program\n\n");
                exit(0);
    }   }
  return 0;
}

// Function to allocate the memory for the struct node
NODE * getnode()
{
  NODE * newnode;
  newnode = (NODE *) malloc(sizeof(NODE));

  // If the memory allocation fails
  // malloc will return NULL
  if(newnode == NULL)
    printf("Memory allocation failed.\n");

  // Return the newnode at any case
  return newnode;
}

void getdata(NODE * newnode)
{
  // Get the information from the user
  // Initialize the next pointer to NULL
  printf("Enter the information of node:\n");
  scanf("%d", &newnode->info);
  newnode->next = NULL;
}

NODE *  insert_at_start(NODE * start)
{
  NODE * newnode;
  newnode = getnode();
  // Memory allocation failed
  if(newnode == NULL)
    return start;
  // Get the data from the user
```

```
   getdata(newnode);

   // If the list is empty, newnode is the start of the list
   if(start == NULL)
      start = newnode;
   else
   {
      // Add the newnode at the beginning and update the start
      newnode->next = start;
      start = newnode;
   }
   // Increment currnodes, print a message and return updated start
   currnodes++;
   printf("%d is inserted at front of the list\n\n", newnode->info);
   return start;
}

NODE *  delete_from_start(NODE * start)
{
   NODE * tempnode;
   if(start == NULL)
      printf("List is Empty!\n");
   else
   {
      tempnode = start;
      start=start->next;

      printf("%d is deleted from front of the list\n\n", tempnode->info);
      free(tempnode);
      currnodes--;
   }
   return start;
}

NODE *  insert_at_end(NODE * start)
{
   NODE * newnode, * nextnode;
   newnode = getnode();
   if(newnode == NULL)
      return start;
   getdata(newnode);

   if(start == NULL)
      start = newnode;
   else
   {
      nextnode = start;
      while(nextnode->next != NULL)
         nextnode = nextnode->next;
      nextnode->next = newnode;
```

```
   }
   currnodes++;
   printf("%d is inserted at the end of the list\n\n", newnode->info);
   return start;
}

NODE *  delete_from_end(NODE * start)
{
   NODE *prevnode, *nextnode;
   if(start == NULL)
      printf("List is Empty!\n");
   else
   {
      if(currnodes == 1)
      {
         nextnode = start;
         start=NULL;
      }
      else
      {
         nextnode = start;
         prevnode = NULL;
         while(nextnode->next!=NULL)
         {
            prevnode = nextnode;
            nextnode = nextnode->next;
         }
         prevnode->next = NULL;
      }
      printf("%d is deleted from end of the list.\n", nextnode->info);
      free(nextnode);
      currnodes--;
   }
   return start;
}

NODE *  insert_at_position(NODE * start)
{
   NODE * newnode, *nextnode;
   int position=0, i=0;

   printf("Enter insert position:\t");
   scanf("%d", &position);
   if(position < 1 || position > currnodes+1)
     printf("Invalid Position!\n");
   else
   {
      newnode = getnode();
      if(newnode == NULL)
         return start;
```

```
      getdata(newnode);

      if(position == 1)
      {
         newnode->next= start;
         start= newnode;
      }
      else if(position == currnodes + 1)
      {
         nextnode=start;
         while(nextnode->next!=NULL)
            nextnode = nextnode->next;
         nextnode->next = newnode;
      }
      else
      {
         nextnode = start;
         i = 1;
         while(i < position-1)
         {
            nextnode = nextnode->next;
            i++;
         }
         newnode->next = nextnode->next;
         nextnode->next = newnode;
      }
      currnodes++;
      printf("%d is inserted at position %d.\n", newnode->info, position);
   }
   return start;
}

NODE *  delete_from_position(NODE * start)
{
   NODE *prevnode, *nextnode;
   int position=0;
   int count = 1;
   if(start == NULL)
      printf("List is Empty!\n");
   else
   {
      printf("Enter node position to delete: \t\t");
      scanf("%d", &position);

      if(position < 1 || position > currnodes)
         printf("Invalid position!\n");
      else
      {
         if(position == 1)
         {
```

```
            nextnode = start;
            start=start->next;
         }
         else
         {
            prevnode = NULL;
            nextnode = start;
            while(count != position)
            {
               prevnode = nextnode;
               nextnode = nextnode->next;
               count++;
            }
            prevnode->next = nextnode->next;
         }

         printf("Node %d with info %d is deleted from List.\n", position, nextnode->info);
         free(nextnode);
         currnodes--;
      }
   }
   return start;
}

void display_list(NODE *start)
{
   NODE * tempnode;
   if(start == NULL)
      printf("List is Empty!\n");
   else
   {
      tempnode = start;
      printf("The list contents are:\n");
      while(tempnode != NULL)
      {
         printf("%d  -->  ", tempnode->info);
         tempnode = tempnode->next;
      }
      printf("NULL\n");
   }
}
```

Doubly Linked List Implementation

```
#include <stdio.h>
#include <stdlib.h>
struct node
{
 int info;
 struct node *next;
```

```
  struct node *prev;
};
typedef struct node NODE;
int currnodes = 0;

// Function prototypes same as singly linked list

int main()
{
   NODE * start;
   start = NULL;
   int choice  = 0;

     .....
    // Will be same as that of singly Linked List
     .....

   return 0;
}

NODE * getnode()
{
   NODE * newnode;
   newnode = (NODE *)malloc(sizeof(NODE));

   if(newnode == NULL)
      printf("Memory Allocation Failed\n");
   return newnode;
}

void getdata(NODE * newnode)
{
   printf("Enetr the information for linked list\n");
   scanf("%d", &newnode->info);
   newnode->next = NULL;
   newnode->prev = NULL;
}

NODE * insert_at_front(NODE * start)
{
   NODE * newnode;
   newnode = getnode();
   if(newnode == NULL)
      return start;
   getdata(newnode);

   if(start == NULL)
      start = newnode;
   else
   {
```

```
      newnode->next= start;
      start->prev=newnode;
      start = newnode;
   }
   currnodes++;
   printf("%d info is inserted at the start of the doubly linked list\n", newnode->info);
   return start;
}

NODE * insert_at_end(NODE * start)
{
   NODE * newnode, *tempnode;
   newnode = getnode();
   if(newnode == NULL)
      return start;
   getdata(newnode);

   if(start == NULL)
      start = newnode;
   else
   {
      tempnode = start;
      while(tempnode->next != NULL)
         tempnode = tempnode->next;

      tempnode->next = newnode;
      newnode->prev = tempnode;
   }
   currnodes++;
   printf("%d info is inserted at the End of the doubly linked list\n", newnode->info);
   return start;
}

NODE * insert_at_position(NODE * start)
{
   NODE *newnode, *tempnode;
   int position=0, i=0;

   printf("Enter insert position\n");
   scanf("%d", &position);
   if(position < 1 || position > currnodes+1 )
     printf("Invalid position\n");
   else
   {
      newnode = getnode();
      if(newnode == NULL)
         return start;
      getdata(newnode);

      if(start == NULL)
```

```
        start = newnode;
      else if(position == 1)
      {
        newnode->next= start;
        start->prev=newnode;
        start = newnode;
      }
      else if(position == currnodes+1)
      {
        tempnode = start;
        while(tempnode->next != NULL)
          tempnode = tempnode->next;

        tempnode->next = newnode;
        newnode->prev = tempnode;
      }
      else
      {
        tempnode = start;
        for(i=2;i<position;i++)
          tempnode = tempnode->next;

        newnode->next = tempnode->next;
        newnode->prev =  tempnode;
        tempnode->next->prev = newnode;
        tempnode->next = newnode;
      }
      currnodes++;
      printf("%d info is inserted at the %d position of the doubly linked list\n", newnode->info,
position);
   }
   return start;
}

NODE * delete_from_start(NODE * start)
{
   NODE * tempnode;
   if(start == NULL)
      printf("List is empty\n");
   else
   {
      if(currnodes == 1)
      {
        tempnode = start;
        start = NULL;
      }
      else
      {
        tempnode = start;
        start = start->next;
```

```
        start->prev = NULL;
      }
      printf("Node %d deleted from the start of the Doubly linked list\n", tempnode->info);
      free(tempnode);
      currnodes--;
   }
   return start;
}

NODE * delete_from_end( NODE * start)
{
   NODE * tempnode, *prevnode;
   if(start == NULL)
      printf("List is empty\n");
   else
   {
      if(currnodes == 1)
      {
         tempnode = start;
         start = NULL;
      }
      else
      {
         tempnode = start;
         while(tempnode->next != NULL)
            tempnode = tempnode->next;

         prevnode = tempnode;
         prevnode = prevnode->prev;
         prevnode->next = NULL;
      }
      printf("Node %d deleted from the end of the Doubly linked list\n", tempnode->info);
      free(tempnode);
      currnodes--;
   }
   return start;
}

NODE * delete_from_position(NODE * start)
{
   NODE * tempnode, *prevnode;
   int position=0, i=0;

   if(start == NULL)
      printf("List is empty\n");
   else
   {
   printf("Enter delete position\n");
   scanf("%d", &position);
   if(position < 1 || position > currnodes )
```

```
      printf("Invalid position\n");
   else
   {
      if(currnodes == 1)
      {
         tempnode = start;
         start = NULL;
      }
      else if(position == 1)
      {
         tempnode = start;
         start = start->next;
         start->prev = NULL;
      }
      else if(position == currnodes)
      {
         tempnode = start;
         while(tempnode->next != NULL)
            tempnode = tempnode->next;

         prevnode = tempnode;
         prevnode = prevnode->prev;
         prevnode->next = NULL;
      }
      else
      {
         prevnode = NULL;
         tempnode = start;
         i = 2;
         while(i<=position)
         {
            prevnode = tempnode;
            tempnode = tempnode->next;
            i++;
         }
         prevnode->next = tempnode->next;
         tempnode->next->prev = prevnode;
      }
      printf("Node %d deleted from the %d position from the Doubly linked list\n", tempnode-
>info, position);
      free(tempnode);
      currnodes--;
   }
 }
   return start;
}

void display_list(NODE * start)
{
   NODE * tempnode;
```

```
   if(currnodes == 0)
      printf("List Empty\n");
   else
   {
      tempnode = start;
      printf("The list contents are:\n");
      printf("\nNULL <--> ");
      while(tempnode != NULL)
      {
         printf(" %d <--> ", tempnode->info);
         tempnode = tempnode->next;
      }
      printf("NULL\n");
   }
}
```

Circular Linked List Implementation

```
#include <stdio.h>
#include <stdlib.h>

struct node
{
   int info;
   struct node *next;
};

typedef struct node NODE;
int currnodes = 0;

NODE *  insert_at_start(NODE * last);
NODE *  insert_at_end(NODE * last);
NODE *  delete_from_start(NODE * last);
NODE *  delete_from_end(NODE * last);
NODE * getnode();
void getdata(NODE *);
void display_list(NODE *last);

int main( )  {
   NODE * last=NULL;
   int choice = 0;
   while(1)
   {
      printf("\n\n* * * * * *   Menu  * * * * * *\n");
      printf("1. Insert node at start\n");
      printf("2. Insert node at End\n");
      printf("3. Delete node from start\n");
      printf("4. Delete node from end\n");
      printf("5. Display List\n");
```

```
        printf("6. Exit\n");
        printf("* * * * * * ******** * * * * * *\n");
        printf("Enter your choice:\n");
        scanf("%d", &choice);

        switch (choice)
        {
           case 1: last = insert_at_start(last);
                 break;
           case 2: last = insert_at_end(last);
                 break;
           case 3: last = delete_from_start(last);
                 break;
           case 4: last = delete_from_end(last);
                 break;
           case 5: display_list(last);
                 break;
           case 6: printf("Exiting program\n\n");
                 exit(0);
        }
     }
     return 0;
}

NODE * getnode()
{
     NODE * newnode;
     newnode = (NODE *) malloc(sizeof(NODE));
     if(newnode == NULL)
        printf("Memory allocation failed.\n");
     return newnode;
}

void getdata(NODE * newnode)
{
     printf("Enter the information of node:\n");
     scanf("%d", &newnode->info);
     newnode->next = NULL;
}

NODE *  insert_at_start(NODE * last)
{
     NODE * newnode;
     newnode = getnode();
     if(newnode == NULL)
        return last;
     getdata(newnode);

     if(last == NULL)
        last = newnode;
```

```
   else
      newnode->next = last->next;

   last->next = newnode;
   currnodes++;
   printf("%d is inserted at front of the circular list\n\n", newnode->info);
   return last;
}

NODE *  insert_at_end(NODE * last)
{
   NODE * newnode;
   newnode = getnode();
   if(newnode == NULL)
      return last;
   getdata(newnode);

   if(last == NULL)
      last = newnode;
   else
      newnode->next = last->next;

   last->next = newnode;
   currnodes++;
   printf("%d is inserted at the end of the list\n\n", newnode->info);
   return newnode;
}

NODE *  delete_from_start(NODE * last)
{
   NODE * tempnode;

   if(last == NULL)
      printf("List is Empty!\n");
   else
   {
      if(last->next == last)
      {
         tempnode = last;
         last = NULL;
      }
      else
      {
         tempnode = last->next;
         last->next = tempnode->next;
      }
      printf("%d is deleted from front of the list\n\n", tempnode->info);
      free(tempnode);
      currnodes--;
   }
```

```
   return last;
}

NODE *  delete_from_end(NODE * last)
{
   NODE *prevnode = NULL;
   if(last == NULL){
      printf("List is Empty!\n");
      return last;
   }
   else
   {
      if(currnodes == 1) {
         printf("%d is deleted from end of the list.\n", last->info);
         free(last);
         currnodes--;
         return NULL;
      }
      else
      {
         prevnode = last->next;
         while(prevnode->next != last)
            prevnode = prevnode->next;

         prevnode->next = last->next;
         printf("%d is deleted from end of the list.\n", last->info);
         free(last);
         currnodes--;
         return prevnode;
      }
   }
}

void display_list(NODE *last)
{
   NODE * tempnode;
   if(last == NULL)
      printf("List is Empty!\n");
   else
   {
      tempnode = last->next;
      printf("The list contents are:\n");
      while(tempnode != last)
      {
         printf("%d  -->  ", tempnode->info);
         tempnode = tempnode->next;
      }
      printf("%d  -->  ", tempnode->info);
   }
}
```

Linked Stack and Linked Queue

A linked stack can be implemented by providing following operations

a) Insert at front and
   Delete from front

b) Insert at end and
   Delete from end

A linked queue can be implemented by providing following operations

c) Insert at front and
   Delete from end

d) Insert at end and
   Delete from front