Paper Name: Data Structures
Unit No : III
Lesson Name: Doubly Linked Lists and Advanced Concepts
Author: Gaurav Saxena, Associate Professor
College/Department: Indraprastha College , University of Delhi

Table of Contents

Chapter 7: Beginning with Doubly Linked Lists

Till last chapter singly linked list and circular linked lists were discussed. For each of them many functions like traversal, merging etc. have been covered. Another variation of singly linked lists is Doubly linked lists which has many advantages over singly linked list and if you read the text carefully enough, you can figure them out all by yourself. Hence, we begin the concept of a doubly linked list with a very easy to grasp and common example that you all have know about.

7.1 Understanding the concept of a doubly linked list.

In simple terms, we can state that a doubly linked list is a list which contains nodes that have at least three fields. One is the information field (there can be any number of information fields) and then there are two link fields. One link field points to the node immediately after this node (Just as in the case of singly linked list) and another link field points to the node immediately behind this node. The former link field can be given a name "next" and the latter link field can be named as "prev". Now, for the sake of clarity we consider the following example.

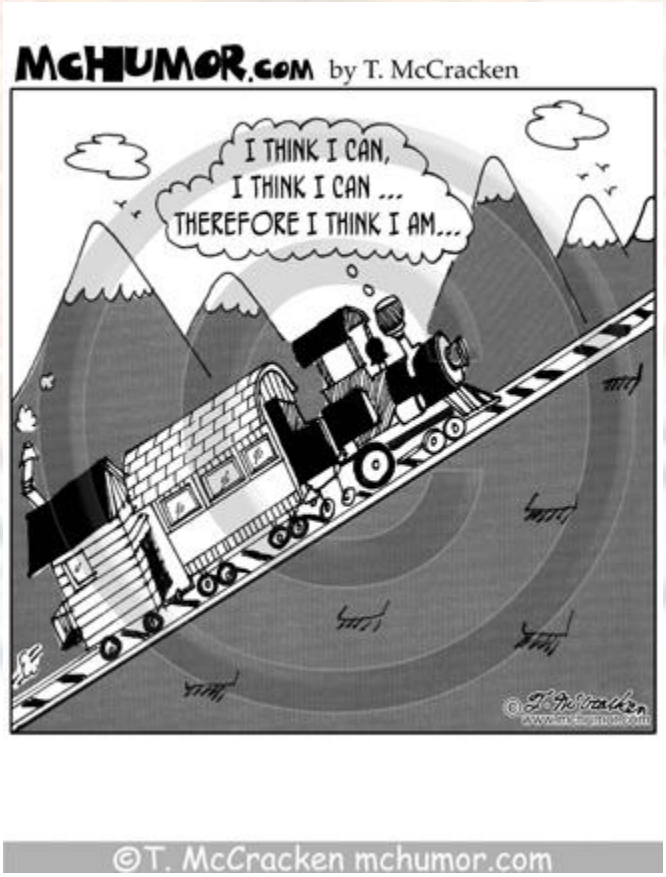


Figure 7.1
http://members.pioneer.net/~mchumor/00images/5047_train_cartoon.gif

Do you know a means of transportation that is of its second largest form in India? Yes, you guessed it right, it is a train. Consider a train with all external doors shut as a doubly linked list with the engine as the first node, the last coach as the last node and all the middle coaches as middle nodes. Now, just as in a train we can move from a coach (a node) to both the next coach (node) and the "prev" coach (node), we can do the same in the doubly linked list.

Keeping this in mind, now you should be able to figure out the following-:

1) Where should the "prev" link of the first node (engine) point?
2) Where should the "next" link of the last node (last coach) point?

If you have been able to visualize a doubly linked list by now, you will be able to figure out that both links in 1 and 2 will be pointing to NULL since no other node(coach) is present before the first node(engine) and after the last node.
It should also be clear that all the remaining nodes (coaches) in between will have both their links as non-NULL.

Value addition: Did you Know
Applications of doubly linked list
1. Applications that have an MRU (Most Recently Used) list (a linked list of file names) 2. The cache in web browser that allows you to hit the BACK button (a linked list of URLs) 3. Undo functionality in Photoshop or Word (a linked list of states) 4. A stack, hash table, and binary tree can be implemented using a doubly linked list. 5. A great way to represent a deck of cards in a game.
Source: Self

7.2 Creating a node of a doubly linked list.

Creating a node of a doubly linked list should not be a problem if you know how to create a node of a singly linked list (explained in previous chapters) and understood the underlying concept that there are two link fields in each node along with its info field.

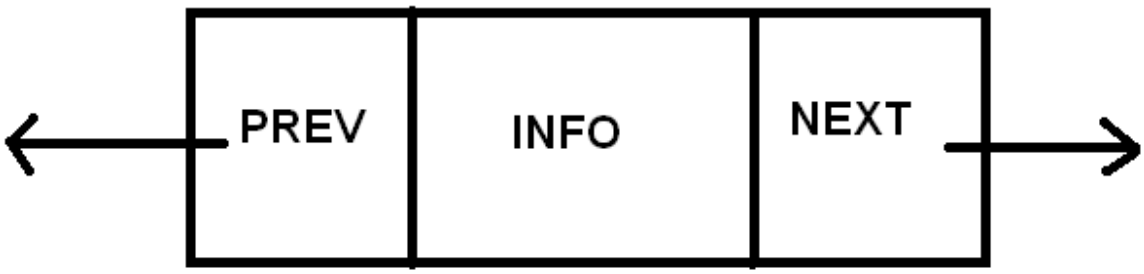


Figure 7.2
Source: Self

7.2.1 Defining the node class

Figure 7.2 shows a typical node of a doubly linked list. We here define a class dnode for creating a node of doubly linked list.

```
class dnode             //class defining a node structure, notice the public instance variables
{
public:
        int info;
        dnode *next;
        dnode *prev;
};
```

Easy enough right?

7.2.2 Defining a doubly linked list class

We now define a class which will define the operations related to a doubly linked list. Again note that this is exactly similar to defining a class for a singly linked list with minor changes that you can figure out yourself if you have grasped the concept above. We are adding comments along with the code for the sake of clarity and to explain the purpose of functions.

```
class dll               //class implementing a doubly linked list
{
private:                //class is private by default, but we can still define it for sake of clarity.
        dnode *p;       // p will point to the starting node.
        public:
        dll();                          //constructor
        void addatbeg(int x);           //add to beginning of list
        void disp();                    //to display the list.
        void revdisp();                 //to display contents in reverse order.
        void addatend(int x);           //add to end of list
```

```
        void addbefore(int x,int y);        //add a node with value x before a node with value y.
        void addafter(int x,int y);         //add a node with value x after a node with value y.

};
```

dll class has a private variable of type dnode* which will act as the pointer to the first node in the linked list – note that until and unless we do not allocate memory to it, it points to some arbitrary memory location at which anything could be present. Also this is just the template – until and unless we create an object of the class "dnode", and make "p" point to it, we cannot proceed. So every time "p" will point to the starting node of a doubly linked list.

7.2.3 Operations on a doubly linked list

We use a constructor dll (), which initializes the value of "p" to be equal to NULL, as the list is empty when it is used the first time. The disp () function, whose work is to simply display the content of the information field in the doubly linked list, is absolutely identical to the singly linked list. Similarly functions like counting the no. of nodes, altering the value of nodes are also exactly similar to that of a singly linked list and are left as an exercise to the reader.

```
dll::dll() //defining constructor outside the class
{
        p = NULL;
}
```

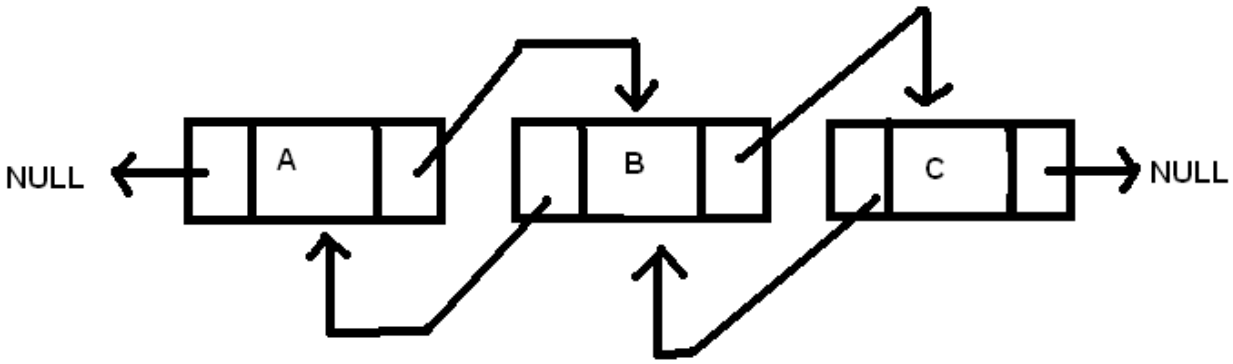


Figure 7.3
Source: Self

Figure 7.3 shows the initial linked list before addatbeg(int x) function.

```
void dll::addatbeg(int x)           //defining function outside the class. Here "x" is the data input.
{
        if(p == NULL)               //If no node initially exists
        {
```

```
                p = new dnode;          //Asking for space to store a node.
                p->info = x;            //Assigning value x to the info field.
                p->next = NULL;
                p->prev=NULL;           //Take into account the "prev" link too(this is the difference)
        }
        else
        {
                dnode *q = new dnode;   //Asking space for a new node
                q->info = x;            // Assigning value of x to the info field.
                q->next = p;            //Same as in SLL.
                q->prev=NULL;           //This step is easy to forget
                p->prev=q;              //Same as above
                p=q;
        }
}
```

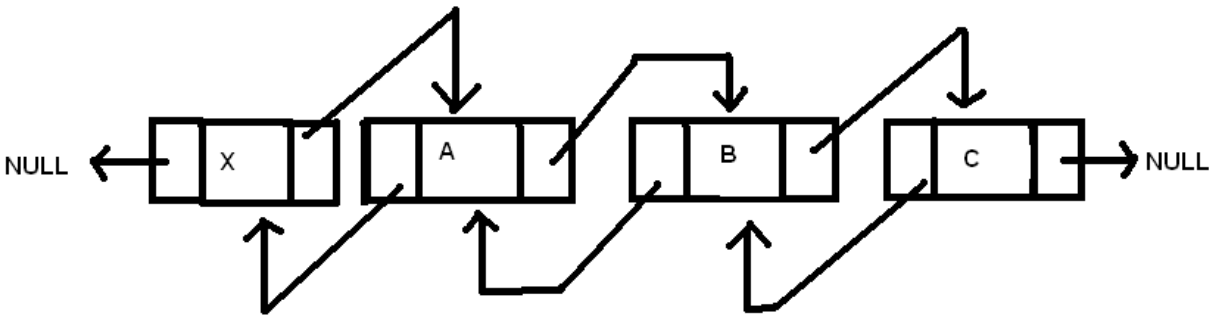


Figure 7.4
Source: Self

Figure 7.4 shows the linked list obtained after addatbeg(int x) function.

Now we write the code for displaying contents of a DLL. To display the content we can either traverse in forward direction or in backward direction.

```
void dll::disp() //Is identical to the one used in SLL
{
 dnode *t = p;

 while(t != NULL)
 {
        cout<<t->info;      //Displays info field
        t = t->next;        //Guess what it does?
 }
}
```

In the function addatbeg (int x), we can clearly see that most of the part here is also same to that of a SLL with a few minor additions. And in most cases these additions are related to assigning the "prev" links of nodes to respective areas. In this function, we are adding a

node at the beginning (visualize it by imagining that you are adding the engine). Here, "q" which points to the new node has its next link pointing to "p"(the first node initially). Now ,we also have to take into consideration the "prev" links of both "p" and "q". There is nothing behind "q" so q->prev should point to NULL. And we are putting "q" before "p" so p->prev should point to "q. Note that it is very common that programmers forget to make the "prev" links of a node point to their proper places. This can be mitigated by visualizing the problem. Try to draw a figure every time you solve a question.

Now we move on to adding a node at the end of a DLL. We first give the code below:-

```
void dll::addatend(int x)
{
        if(p == NULL)                       //if DLL is empty
        {
                p = new dnode;
                p->info = x;
                p->next = NULL;
                p->prev=NULL
        }
        else
        {
                dnode *t = p;               //make a copy of "p". Why?
                while(t->next != NULL)      //traverse till you reach the last node
                        t= t->next;
                dnode *q = new dnode;
                q->info = x;
                q->next = NULL;
                q->prev=t;                  //Notice this step
                t->next = q;
        }
}
```

The function addatend(int x): In this function also almost everything is the same as add a node in the end of a SLL except for one step i.e. here we also make the "prev" field of newly made node "q" point to the previous node(t in this function). And that is it. We hope functions like these do not cause much of a problem.
Now we move on to the function that displays the contents of a DLL in reverse order. Here goes the code:

```
void dll::revdisp()
{
 dnode *t = p;

 while(t->next != NULL)          //go till the last node
 {
          t = t->next;           //Guess what it does?
 }

 while(t ! =NULL)                // till t becomes null
{
   cout<<t->info;                // display the info
   t=t->prev;                    //move backwards (this is the advantage of dll)
```

```
}
}
```

In the above function, our objective is to print the DLL backwards. Here we can use the "prev" link of each node to our advantage. We first go till the end of the linked list by using the "next" link field, and then print info field while moving backwards by using the "prev" link of the last node and so on. This is wherein the power of a DLL lies. And here we use it to our advantage and we will keep using it in many more function to come. Imagine a recursive function in SLL to print a list backwards. Which one do you find easier?
Now we explain a function to add a node before a node that contains the value y. the code will be similar to that of an SLL, but we would just need to take care of the "prev" field of the node to be added and the nodes immediately after it (i.e node with value y).

```
void dll::addbefore (int y,int x)
{
        dnode *t = p;

        while(t->info != y && t->ptr != NULL) //stops when last node is reached
        {
                t = t->ptr ;
        }

        if(t->ptr == NULL && t->info != y)     //checking for last node
                cout<<" Value "<<y<<" does not exist "<<"\n";
        else                            //node with value y is found
        {
                dnode *q = new dnode;
                q->info = x;
                q->next = t;
                q->prev=t->prev;  //notice the step carefully.will it work if t is 1st node? Yes.

                 if(p->info!=y) // Checking the first node for value y
                        t->prev->next=q;  //this comm. will crash prog. if applied on 1st node.
                 else
                        p=q;
                t->prev=q;
        }
}
```

Adding a node before a given node in DLL is similar to SLL. In the above function, everything is just the same as the function in SLL, with some subtle changes. As we know, in SLL we need to maintain a pointer to stay one step behind the node with value y. Try to figure out as to why this is needed. Since DLL's nodes have a link to the previous node, we can refer to the previous node indirectly using the former. This is exactly what we have done by using command t->prev->next=q. This statement makes the "next" field of the node initially before the node with value y point to our new node. Note that this command can cause serious consequences if applied on 1st node. Why? Because then we will be referring to the NULL pointer that will crash the program i.e. produce a run time error. On a similar basis we put the "prev" link field of "q" point to this initially behind node with the command q->prev=t->prev. Try to draw a figure to understand proper linking over here.

Now moving on, we give the code for adding a node after a node with value y. this function is simpler than the previous one. We provide comments to make the code self explanatory.

```
void dll::addafter(int y,int x)                    //adds value x after given value y
{
        dnode *t = p;                              //Creating a copy of "p"
        while(t->info != y && t->ptr != NULL)
                t = t->ptr ;

        if(t->ptr == NULL && t->info != y)
                cout<<" Value "<<y<<" does not exist "<<"\n";
   else
        {
                dnode *q = new dnode;
                q->info = x;
                q->prev=t;
                q->next = t->next ;
                if(t->next !=NULL)                 //only if it is NOT the last node
                t->next->prev=q;
                t->next = q;
        }
}
```

Intially we check if a node with value y exists (trivial case) or not. If it does then we create a node, put its info as x and reassign link fields. We make the "next" field of new node(q) point to the node pointed by "next" field of "t". We make the "prev" link field of "q" point to t itself. Next we check if the node with value y is the last node in the list because then we do not have to worry about making the "prev" link field of successive node point to q(since it doesn't exist). If "t" is not the last node then we make the "prev" field of node initially next to "t" point to "q" by the command t->next->prev=q.

Value addition: Did you Know

GENERAL TIPS:

1) Whenever you are adding a node in a doubly linked list before/after some node, you will have to reassign exactly four link fields. Two of the newly created node, and the "next" field of the node initially immediately behind the target node ("t" in previous example) and the "prev" field of the node initially after the target node.
2) Check for condition of the target node before writing any command that can lead to system crash. For example, in the function to add a node after a target node, check if the target node is the last node and then only perform something like t->next->prev. If you forget you may be unintentionally playing with the NULL pointer that can crash your program.
3) Always draw figures to visualize the situation. That will make the logic for writing the code clearer.
4) In the subsequent programs, you may want to maintain a pointer to the end of the doubly linked list as this may save your time to traverse the whole list to reach the end node to perform operations like displaying the contents in reverse etc.

Source: Self

Value addition: Did you Know
Doubly Linked List vs. Singly Linked Lists
Double-linked lists require more space per node, and their elementary operations are more expensive; but they are often easier to manipulate because they allow sequential access to the list in both directions. In particular, one can insert or delete a node in a constant number of operations given only that node's address. To do the same in a singly-linked list, one must have the *previous* node's address. Some algorithms require access in both directions. On the other hand, they do not allow tail-sharing, and cannot be used as persistent data structures.
Source: Self

Chapter 8: Doubly Linked Lists – Advanced usage and concepts

This chapter is devoted to some advanced applications and functions of doubly linked lists that include reversing a doubly linked list, performing binary search on it, deletion of nodes, finding its middle node. Finally, we discuss its application in implementing a matrix.

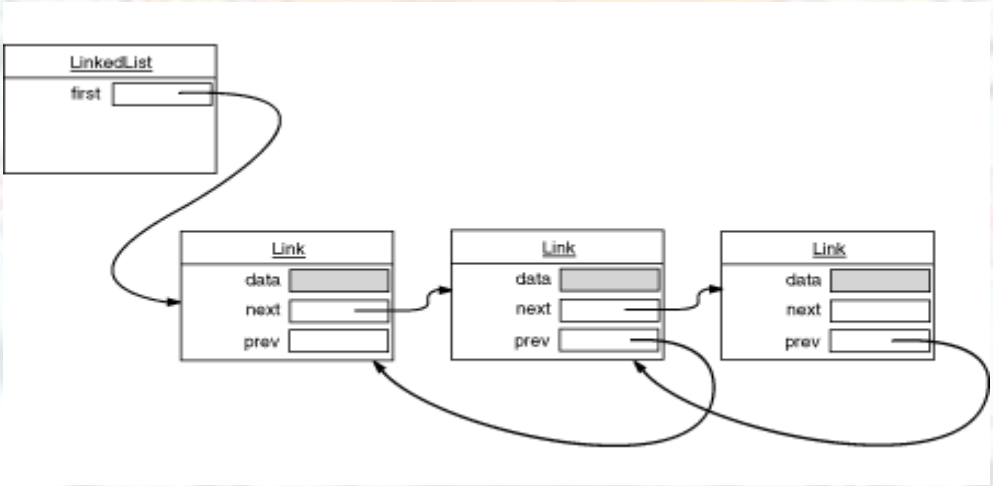


Figure 8.1
http://java.sun.com/developer/Books/javaprogramming/corejava/Fig2-05.gif

8.1 Reversing a doubly linked list

Implementing a reverse operation on a doubly linked list might seem difficult at first when compared to a singly linked list but in fact it is easier as a node of a doubly linked list contains links to both the previous node and the next node. The logic is relatively easy. All

we have to do is swap the next and prev link fields of each and every node, and in the end ,point the starting pointer at the last node. You can draw a diagram to verify the logic.
Here is the code:

```
void dll::reverse()
{
  dnode *q=p;
  dnode * s=p,*temp;

  while(q!=NULL)
    {
       q=q->next;
       temp=s->next;        //swapping link fields
       s->next=s->prev;
       s->prev=temp;
       if(q!=NULL)
        s=q;                 // Move to next node
     }
    p=s;
}
```

In the above function, we first make two copies of the starting pointer (i.e. q and s). we run the loop till q equals NULL or in other words the end of the linked list has been reached. In the while loop, first statement advances q to the next node. Pointer s still points to the previous node. Now we swap the next and prev link fields of the node pointed to by s and then move on to the next node by assigning s the value of q and this goes on and on and on. So are we taking care of the trivial cases? What if linked list is empty? What if the linked list contains a single node only? This is left as an exercise to the reader. In the above function again we saw the huge advantage of dealing with doubly linked lists.

Value addition: Did you Know

Two other techniques for reversing a Doubly linked list

There are a couple of other ways to reverse a linked list.

Technique 2:
In this way, a new linked list will be created and all the items of the first linked list will be added to the new linked list in reverse order.

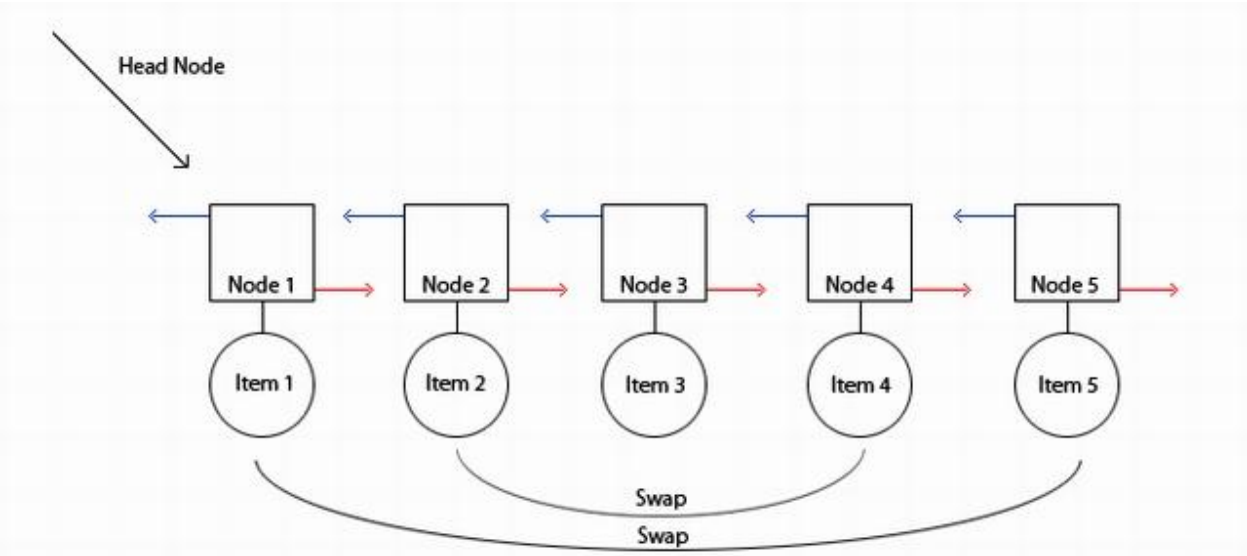
```
public void ReverseLinkedList (LinkedList linkedList)
{
// ------------------------------------------------------------
// Create a new linked list and add all items of given
// linked list to the copy linked list in reverse order
// ------------------------------------------------------------

LinkedList copyList = new LinkedList();

// ------------------------------------------------------------
// Start from the latest node
// ------------------------------------------------------------
```

```
LinkedListNode start = linkedList.Tail;

// ------------------------------------------------------------
// Traverse until the first node is found
// ------------------------------------------------------------

while (start != null)
{
// ------------------------------------------------------------
// Add item to the new link list
// ------------------------------------------------------------

copyList.Add (start.Item);

start = start.Previous;
}

linkedList = copyList;

// ------------------------------------------------------------
// That's it!
// ------------------------------------------------------------
}
```

This way is probably the most inefficient among the three. Even though objects of each node are not copied to memory, a new link list node is created for each call to "Add" property. A new link list node means at least 3 pointers (Next, Previous, Item) will be created for each item. This method not only wastes memory, but also wastes processing power.

Technique 3:

In this method, we will swap linked list node objects (references to the data). Swapping starts from the first node's object and the first node's object is swapped with the last node's object. Then, the second node's object is swapped with the one before the last node's object.



Swapping goes on until the middle of the linked list is found.

```
public void ReverseLinkedList (LinkedList linkedList)
{
// ------------------------------------------------------------
// Create variables used in the swapping algorithm
// ------------------------------------------------------------

LinkedListNode firstNode; // This node will be the first node in the swapping
LinkedListNode secondNode; // This node will be the second node in the swapping
int numberOfRun; // This variable will be used to determine the number of swapping runs

// ------------------------------------------------------------
// Find the tail of the linked list
// ------------------------------------------------------------

// Assume that our linked list doesn't have a property to find the tail of it.
// In this case, to find the tail, we need to traverse through each node.
// This variable is used to find the tail of the linked list
LinkedListNode tail = linkedList.Head;          // Start from first link list node
                                                // and go all the way to the end
while (tail.Next != null)
tail = tail.Next;

// ------------------------------------------------------------
// Assign variables
// ------------------------------------------------------------

firstNode = linkedList.Head;
secondNode = tail;
numberOfRun = linkedList.Length / 2;
// Division will always be integer since the numberOfRun variable is an integer
```

```
// ------------------------------------------------------------
// Swap node's objects
// ------------------------------------------------------------

object tempObject; // This will be used in the swapping 2 objects
for (int i=0; i < numberOfRun; i++)
{
// Swap the objects by using a 3rd temporary variable
tempObject = firstNode.Item;
firstNode.Item = secondNode.Item;
secondNode.Item = tempObject;

// Hop to the next node from the beginning and previous node from the end
firstNode = firstNode.Next;
secondNode = secondNode.Previous;
}

// ------------------------------------------------------------
// That's it!
// ------------------------------------------------------------
}
```

This way of reversing a linked list is pretty optimized and pretty fast. The only overhead of this algorithm is finding the end of the linked list.

But as you will realize that the best method for implementing a reverse operation is the one that we discussed in the core text as reversal takes place in a single pass only with no tail pointer.

This example was illustrated that you can develop many ways of implementing any operation provided the concept is clear!

Note: This is a quoted example so the coding syntax is different. Reader is requested to grasp the concept of reversal rather than focusing too much on the code!

Source:
http://www.codeproject.com/KB/recipes/ReverseLinkedList.aspx

8.2 Finding the middle node of a doubly linked list

Well, this function does not only apply to a doubly linked list, it also applies to a singly linked list. Finding the middle node of any list is beneficial in many ways, one of them being able to find the average using median method. If you get a thorough hold of this concept, you can assure yourself of finding almost any node of any linked list! Let us put the thinking cap on. So how do we go about this problem? For a doubly linked, finding a suitable logic is easy.

We can make a copy of the starting pointer and we can traverse to the end of the list a make a pointer point to its tail (if not already present). We then advance the starting pointer forward and tail pointer backwards one at a time simultaneously till they meet.The node at which they meet will be the middle node. But is this code efficient? Well let's face it! This code only applies to doubly linked list and is a little complex. A better idea would be to make two copies of the starting pointer. Advance one pointer one node at a time and advance one pointer two nodes at a time. Any guesses where the solution would lie? Imagine a hero and a villain in a burning train fighting in the engine. Well unfortunately it so happens that the girl is at the end of the train. The hero (obviously runs twice the speed to

that of the villain) decides to run save the day. Villain runs too at the same time to ruin his plans (Visualize this example in context of a SLL also).

Hero reaches the end of the train successfully (like in most Bollywood movies), now simple math can tell us that the villain would have only covered half the train when hero would have reached the end, or in other words, villain would have reached the middle. Same is the inspiration that we put to use here. When the node that is covering two nodes at a time reaches the end, the node that was covering one node at a time would reach the middle node and hence the solution. A problem here is that a linked list with even nodes would have two nodes as its middle part. It is important here to understand that if a pointer is jumping two nodes at a time in a linked list then at one point, node which it is at currently, that node's->next->next field will be NULL (draw a diagram to approve of this). And if the list is odd then node's->next field will be NULL. We use this distinction to know if the linked list is odd or even and manage the middle point accordingly.

Value addition: Did you Know

Code for finding the middle node (applies to SLL also)

```
void dll::midnode()
{
   dnode *q=p, * s=p,*temp;;

 if(p==NULL)
  {
    cout<<"list is empty"<<'\n';
    return;
   }

while(q->next!=NULL || q->next->next!=NULL)
    {
       q=q->next->next;
       p=p->next;
     }

   if(q->next==NULL)                                    //List is odd
    cout<<"The info in the mid node is"<<p->info;       //clearly, p is the middle node
  else                                  //List is even (p and the node after p are the middle nodes)
    cout<<"The info in the mid nodes is "<<p->info<<" and"<<p->next->info;
 }
```

Source: Self

If the concept for implementing above function is clear then you can try the following exercise: Find the node which cuts the linked list in the ratio of 1:3. And write an appropriate code for it. Now we move on to deletion of nodes from a doubly linked list.

8.3 Deleting nodes in a doubly linked list

Nodes can be deleted in two ways as done before in SLL. i.e. by value or by position. First we will see deletion of nodes by position. Concept remains same as that of deleting nodes in

a singly linked list with only difference of handling an extra pointer 'prev'. Here we give the code for deleting a node with position specified by the user.

```
void dll::del1(int x) //here x is the position
{
        int totalcount = 0; //to count no. of nodes
        int count;
        dnode *t = p;   //t will be the target node

   while(t != NULL)
        {
                t = t->next ;
                totalcount++;     //counting total no. of nodes
        }

        if(x > totalcount)
             cout<<"You specified an invalid position";  //trivial case
   else
        {
                if(x != 1 && x!=totalcount)    //node to be deleted is neither the first node nor the last node
                {
                        count = 0;
                        t = p;

                        while(count <= x-1)   //reaching the target node, since count is beginning with 0.
                        {
                                t = t->next;
                                count++;
                        }
                        t->prev->next = t->next;   //Notice and understand these steps using diagrams
                        t->next->prev=t->prev;
                        delete t;
                }
                else if( x==1) //node to be deleted is the first node
                {
                        dnode *q = p;
                        p = p->next;
                        p->prev=NULL;    //the prev field of second node has to be NULL.
                        delete q;
                }

                else  //node to be deleted is the last node
                { t->prev->next = NULL;
                  delete t;

                 }

        }

}
```

As, you can see if the logic is clear, deleting node from a DLL, is easy if the concept is clear enough. First we check for the trivial case, i.e. if the position specified by the user exists or not. Then we go to that position and make the next field of the node that is previous to the

target node point to the next node of the target node. Similarly, we make the prev field of the node that is next to the target node point to the previous node of the target node and then we delete the target node. Of course this is only possible if the target node is NOT the first node of the list or the last node of the list and in these cases appropriate commands have been given. Will this function work if the linked list is empty?

Now we give the code for deleting a node by value. Its code has intentionally been kept different from the earlier codes to develop the understanding skills of the reader.

```
void dll :: d_delete ( int num ) //num is the value specified here
{
        dnode *q = p ;

        // traverse the entire linked list
        while ( q != NULL )
        {
                // if node to be deleted is found
                if ( q ->info  == num )
                {
                        // if node to be deleted is the first node
                        if ( q == p )
                        {
                                p = p -> next ;
                                p -> prev = NULL ;
                        }
                        else
                        {
                                // if node to be deleted is the last node
                                if ( q -> next == NULL )
                                        q -> prev -> next = NULL ;
                                else
                                // if node to be deleted is any intermediate node
                                {
                                        q -> prev -> next = q -> next ;
                                        q -> next -> prev = q -> prev ;
                                }

                        }

                  delete q;
                  // return back after deletion
                        return ;
                }

                //if match is not found, go to next node
                q = q -> next ;
          }
        cout << "\n" << num << " not found." ; //It will come here if no match is found.
 }
```

Above function executes the deleting operation in a single while loop. The code has been supplemented with suitable commands to make the reader clear of the logic. Here we discuss it briefly. The while loop runs till the pointer reaches the end of the linked list (i.e. it becomes NULL). In the while loop it tries to find a match with every node. If the target node is found, then it checks whether the node is the first node or the last node or any other

intermediate node. If the node is the first node, starting pointer p is advanced one node and its previous link is pointed to NULL. And the initial node (q) is deleted. If the node is the last node, then next field of the node previous to this node is made to point to NULL and this node is deleted. If the node is intermediate, identical steps are performed as they were in the previous function. Now we move on to Circular doubly linked list.

8.4 Circular doubly linked list

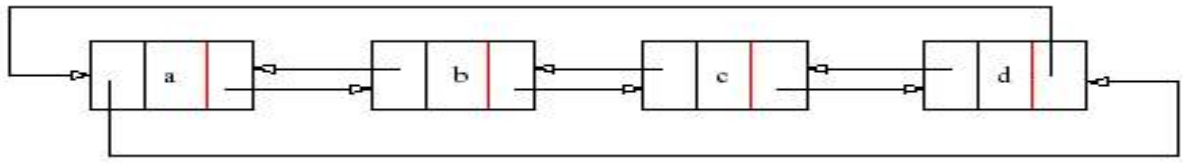


Figure 8.2
http://www.mec.ac.in/resources/notes/notes/ds/ll_files/image018.jpg

Figure 8.2 shows a circular doubly linked list.

Circular doubly linked list are a unique implementation of doubly linked list in which the last node of the list points to the first node of the list.
Clearly, to implement a circular doubly linked list, you have to make the last node should point to the first node q->next=p where q is the last node. Then if we are traversing the list, what will be the terminating condition? For testing the end of the list while traversing, we write.

```
if(q->next==p)
        cout<<" End of linked list reached";
```

Value addition: Did you Know
General Tips:
While implementing circular doubly linked lists we have to keep the following things in mind-: 1) Terminating condition of linked list has changed. 2) The prev field of starting pointer no longer points at NULL. It points to the last node instead. The last node of the linked list does not point to NULL any longer. It points to the first node in the list. Hence no node has a NULL value in its pointer field.
Source: Self

Now we present the code for performing various operations on a CDLL.

8.4.1 Displaying the CDLL

Just keep the terminating condition in mind and displaying a CDLL is a piece of cake.

```
void dll::display()
{
  dnode *traverse = p;

  if(p != NULL)
  {
    while(traverse->next != p)
            {
              cout<<traverse->info<<"-->";
              //cout<<"("<<traverse->prev->info<<")";
              traverse = traverse->next ;
            }
              cout<<traverse->info<<"-->";
              cout<<"CirclesAround-->";
              cout<<traverse->next->info ;
     //cout<<"("<<traverse->prev->data<<")";
  }
}
```

This code is quite clear.

8.4.2 Adding a node at the end of the CDLL

This operation code is also quite easy.

```
void dll::aaddatend (int num)
{
            dnode *q;

            if(p == NULL)
            {
                        p = new dnode;
                        p->info = num;
                        p->prev = p;
                        p->next = p;
            }
            else
            {

                        q = new dnode;
                        q->info = num;
                        q->prev = p->prev;
                        q->next = p;
                        p->prev = q;
            }
}
```

As you can see the code for adding a node at end of a CDLL is quite similar to adding a node after a node in a DLL. (Consult addafter () in chapter 7).

8.4.3 Deleting a node from a CDLL by value

```
void dll::d_delete(int num) //num is the value given
{
   if(p->info == num)  //if first node contains the value
          {
            dnode *temp = p;
            p = p->next;
            temp->prev->next = p;
            p->prev = temp->prev ;
            delete temp;
            return;
          }

          if(p->prev->data == num)
          {
             dnode *temp = p->prev;
             temp->prev->next = p;
             p->prev = temp->prev;
             delete temp;
             return;
          }
          else
          {
                    dnode *traverse = p;

                    while(traverse->data != num)
                              traverse = traverse->next;

                    dnode *temp = traverse;

                    traverse = traverse->prev;
                    traverse->next = temp->next ;
                    temp->next->prev = traverse;
                    delete temp;
                   return;
          }
       else
      cout<<"Node not found"<<'\n';
}
```

Go through the above code carefully and understand it with the help of diagrams.

Value addition: Did you Know
Circular linked lists vs linear linked lists
A circularly linked list may be a natural option to represent arrays that are naturally circular, e.g. for the corners of a polygon, for a pool of buffers that are used and released in FIFO order, or for a set of processes that should be time-shared in round-robin order. In these applications, a pointer to any node serves as a handle to the whole list. With a circular list, a pointer to the last node gives easy access also to the first node, by following one link. Thus, in applications that require access to both ends of the list (e.g., in the implementation of a queue), a circular structure allows one to handle the structure by a single pointer, instead of two. A circular list can be split into two circular lists, in constant time, by giving the addresses of the last node of each piece. The operation consists in swapping the contents of the link fields of those two nodes. Applying the same operation to any two nodes in two distinct lists joins the two lists into one. This property greatly simplifies some algorithms and data structures, such as the quad-edge and face-edge. The simplest representation for an empty circular list (when such thing makes sense) has no nodes, and is represented by a null pointer. With this choice, many algorithms have to test for this special case, and handle it separately. By contrast, the use of null to denote an empty linear list is more natural and often creates fewer special cases.
Source: Self

8.5 Binary search tree

First of all what is a tree, a tree is an object having a root, a branch and some leaves.
Now consider a tree put upside down beginning with the root and going down till its leaves.
In a similar way consider an arrangement of doubly linked list nodes similar to this tree with the condition that a node cannot have more than two links. Instead of using next and previous fields, we use right and left as field names. We can emulate a binary tree using doubly linked list in the following way-:

1) First there is the root (or the starting node). There is no node above this node. Also this node can have a maximum of two links that point to its children below, i.e. the left child and the right child.

2) In a similar fashion, the left child node can have a maximum of two links that point to it left child and right child. In case there is no left child, then the left link field points to NULL. It may not have a right child, in that case the right link field points to NULL. It may also not have either child, and in that case, both the link fields point to NULL. 3rd case is the special case and a node satisfying it is known a leaf node (terminating node).

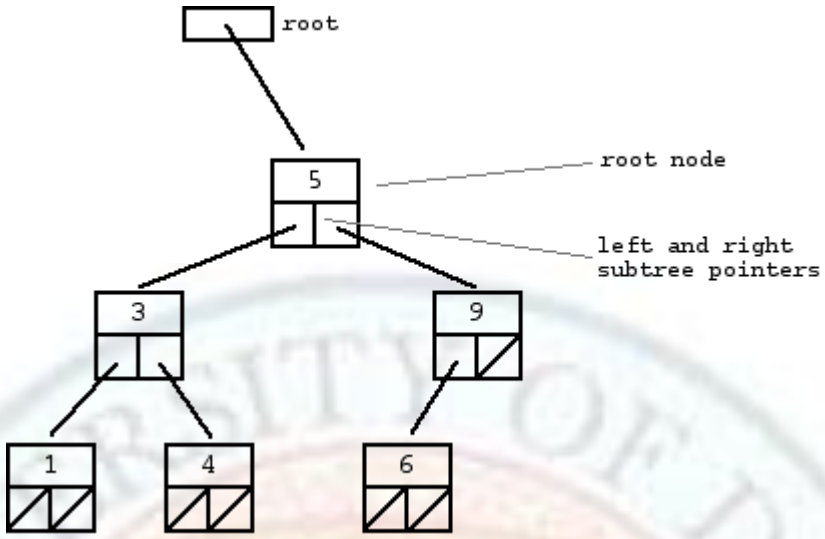


Figure 8.3
http://cslibrary.stanford.edu/110/binarytree.gif

Figure 8.3 shows a binary tree with nodes. Note that each node has two link fields .One points to the left and one to the right. Link fields with a '/' indicate that they are NULL pointers.
We can clearly see the root node and the leaf nodes (the terminating nodes).
When there is a condition that a node cannot have more than two children, the tree is known as a binary tree. And when there is no such restriction, the tree is known as a multi-way tree as shown in figure 8.4

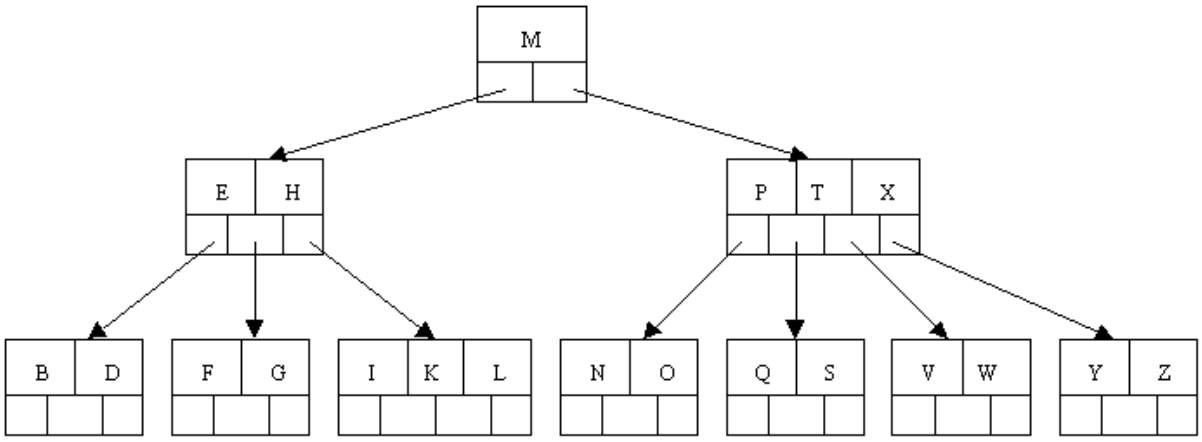


Figure 8.4
http://cis.stvincent.edu/html/tutorials/swd/btree/btree1.gif

Multi-way tree is beyond the scope of this chapter and you will study it in higher classes.
Now we consider a special case of a binary tree that is a binary search tree.

Binary search tree

Binary search tree is tree in which the nodes are inserted on basis of their information part. The typical convention is that if the information value of a node is less than the value of its parent, then it becomes its left child and if the information value is greater than it becomes a right child.
Therefore, the following conditions are satisfied-

- The left sub-tree of a node contains only nodes with keys less than the node's key.
- The right sub-tree of a node contains only nodes with keys greater than or equal to the node's key.
- Both the left and right sub-trees must also be binary search trees.

Value addition: Did you Know

Binary search tree example

Consider making a binary search tree when nodes with following value are inserted: 38,13,51,10,12,40,84,25,89,37,66,95

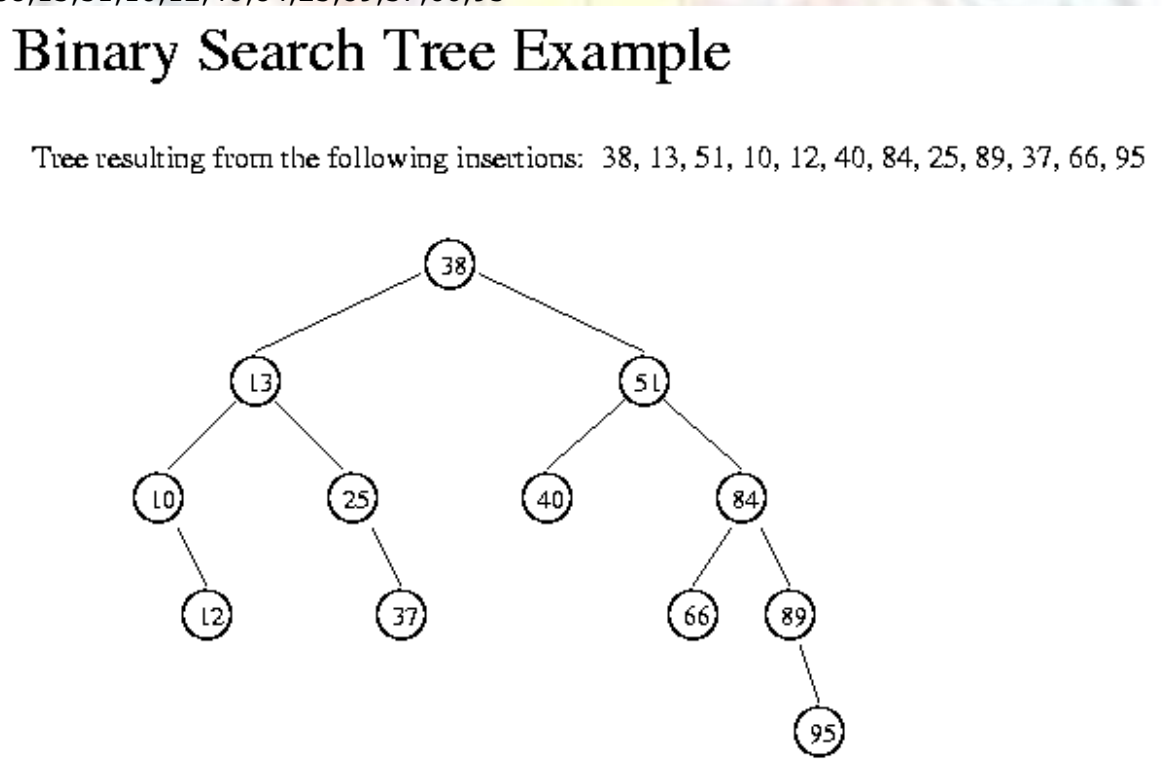


Figure 8.5
http://www.cosc.canterbury.ac.nz/tad.takaoka/alg/dict/bst.gif

First 38 gets inserted since there is no other node. Then 13 is taken and compared with 38. Since it is less than 38, it becomes 38's left child. Next comes 51 and being greater than 31,it becomes the right child. Similarly 10 is less than 38 and less than 13 too so it becomes the left child of 13. 12 is less than 31, less than 13, but greater than 10. So it becomes the right child of 10. And it goes on and on like this. Reader is advised to draw the tree and perform this herself/himself.

Source: Self

Chapter 9: Miscellaneous issues about SLL and CSLL

In this chapter we will discuss topics like finding loops in any list, and discuss about increasing efficiency of a linked list by using skip lists and self organizing list. We will also give a short description about stacks and queues.

9.1 Finding loops in a linked list

In commercial software, where sometimes huge linked lists are implemented which contains sensitive data, it is easy for the data to get attenuated and corrupted by various means like physical corruption, human error, hacking etc. So in the long run, it is important that we are able to diagnose the problems in a linked list. One of these problems is that how to detect ill formed loops in a linked list. By loops in a linked list we mean that any node in a linked list may point to the other node that it is not supposed to point to. Look at the figure below. It shows a node in a SLL that is illegally pointing to another node.

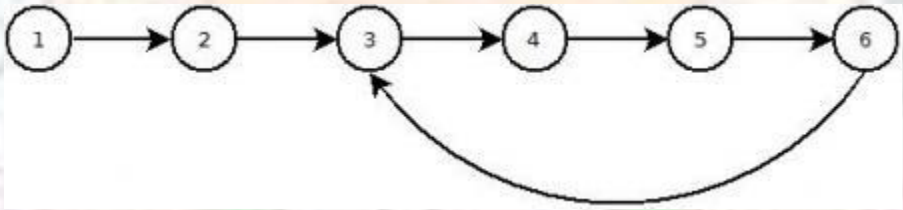


Figure 9.1
Source: Self

Having loops like this can wreck havoc on any program. So, it is essential that we realize how to detect loops in a linked list. Approach to detecting such loops is simple and follows:
Take two pointers from the starting position, one that traverses two nodes at a time and a one that traverses one node at a time. Here the logic is that if the linked list is circular (i.e. there is a loop in the linked list), then these two pointers will point reach the same node at the same time. You can apply this on the figure above and feel sure that this technique will always work, even if there is only a single node in the list. If you dry run the logic on the given figure 9.1 you will find that the two pointers will meet each other again on node with value 5.

Value addition: Did you Know
Code for finding a loop in a linked list

Below we give you the code for implementing this logic.
void list::loop()
{
node * q=p; // q covers two nodes at a time
node * t=p; // t covers one node at a time

```
  while( q!=NULL || q->ptr!=NULL)          //if it reaches this condition is falsified, then there is no
loop
 {
    q=q->next->next;
    t=t->next;
    if(t==q)                               //if they point to same node ,then there is a loop in the
list.
    {
      cout<< " There is a loop in a list"<<'\n';
      getch();
      return;
     }
 }
 cout<<"There is no loop in the list"<<'\n';
}
```

Explanation is avoided for the above code since the comments are self-explanatory.

Now when we have detected the loop in the linked list, the question arises that how do we fix it. Note that above code does not tell us which node's link is corrupted. So we have to find that node and change its link. Finding the node is relatively easy. One logic can be to make a copy of the starting pointer and make it to traverse the list one by one. At each traversal, calculate the length of the list from starting pointer to that node. Also make a pointer (let it be t) that stays one node behind the previous copy. Store the length in a variable and check its value after each iteration. If the value of the variable decreases (i.e. the length decreases) then it means that we have reached back somewhere in the list and so the faulty node will be pointed to by t. And then we can correct the link of that node to point to the correct node as desired in that situation.

Source: Self

9.2 Stacks

A Stack is a data structure that implements the technique of FILO (i.e. the value that goes in first when stack is filled comes out last when the stack is emptied) or LIFO (Last in first out). Consider a heap of dishes and suppose that you have to arrange them in a vertical column. You pick up the first dish and keep it down, and then you pick up the second one and keep it over the top of the first one and so on till there are no more. Figure 9.2 illustrates this concept.

There are two basic operations performed on stacks: push and pop. Figure 9.2 shows and example of a stack of dishes. From this stack, if you pick up the first dish, it would be the last dish that you had put.
And this is exactly how stacks operate. Stacks can be implemented using any data structure like array or a linked list but here we will discuss the concept using a singly linked list.
To form a stack you add nodes at the beginning of a linked list one at a time.
Similarly, delete nodes at the beginning to implement the pop operation.

The push operation inserts onto a stack a particular value at the top position. The pop operation takes away from the stack the topmost value. In real life when the memory of a

computer system is limited it is possible that when we push an element no memory can be allocated to it hence we encounter the overflow error. When the stack is empty and we pop an element what we get is an underflow error. Notice that theoretically the overflow operation can be removed when the memory in the system is made infinite. But as far as the underflow operation is concerned it cannot be removed by any theoretical constraint. Hence in theory only the underflow error exists and never the overflow operation. These two operations can be combined to give several new operations like a MULTIPUSH (s, k) which is pushing k elements at one time on top of stack s, seetop () which is to examine the topmost element without actually removing it hence consists of popping the element and pushing it back, MULTIPOP(s, k) which is popping k elements out of stack s. The last operation if you notice does not necessarily pop out k elements when the number of elements on the stack is less than k.

Figure 9.2
http://images.inmagine.com/img/corbis/crbs079/crbs079112.jpg

To implement a stack, only two operations are required. One is to add the nodes at beginning. This operation is known as push. And the other one is to remove the nodes from the beginning. This operation is known as pop. Please note that these names are used in proper terminology so it is advised that readers must use these names when implementing a stack. Note that delete is not the correct term to be used here; obtaining an element from a stack is far better a statement. These functions are absolutely identical to the ones mentioned in previous chapters which went by the name of addatbeg () and deleteatbeg (). Note that you can ONLY use these two functions when using a stack since no other function can be practically performed on a stack and hence they are restricted. Also note that the pointer p that previously used to denote the starting pointer will now be called as top of stack.

Value addition: Did you Know

Code for performing push and pop operations on a stack

```
void push(int x) //Here "x" is the data input.
{
        if( top == NULL)                    //if the stack is empty
        {
                top = new node;             //Asking for space to store a node.
                top->info = x;              //Assigning value x to the info field.
                top>ptr = NULL;
        }
        else
        {
                node *q = new node;         //Asking space for a new node
                q->info = x;                // Assigning value of x to the info field.
                q->ptr = top;               //Assigning the address of older top to link field.
                top = q;            //This node has now been created and top now points to it.
        }
}

int pop()
{
         if( top == NULL)                   //if the stack is empty
        {
                cout<<"Underflow"<<'\n';
                return -1;
          }
        else
        {
                node * q;
                 return (top->info); //we return the value of top to the program
                q=top;
                top=top->ptr;       //we advance top one step.
                delete q;           //we delete the earlier node pointed by top
         }
}
```

Source: Self

Figure 9.3 illustrates this.

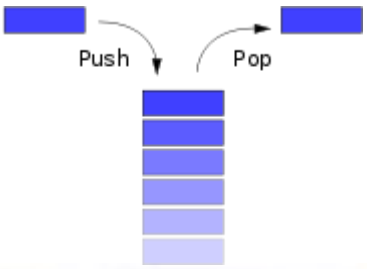


Figure 9.3
http://upload.wikimedia.org/wikipedia/commons/thumb/2/29/Data_stack.svg/180px-Data_stack.svg.png

One application of stack is the system stack that is utilized when recursive procedures are being called. Another application is to add very large numbers as discussed before using linked lists.
Stacks are also used to check the correctness of a code for Example to check expression validity. Here we push each opening bracket in an equation and pop them out when a closing bracket is encountered. If after the end of equation, and bracket is left in the stack, the equation is deemed incorrect.
Now we leave you with an exercise -: If you push the letters 't' , 'e' , 'a' , 'c' , 'h' in a stack and pop them at end, what will be the output? What if you push and pop each letter simultaneously? How many combinations of these words can you get by using a stack?

9.2 Queues

A queue is data structure which implements FIFO (First element in is the first element out) or LILO (last element in is the last element out). An example could be a queue of people at a milk booth, the person who was first in line will be the first one to get the milk and come out.

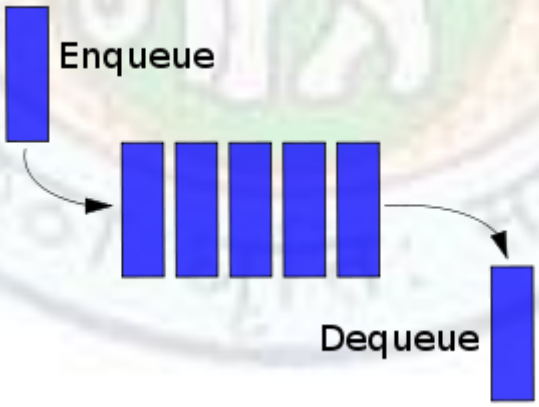


Figure 9.4 (a)
http://stackoverflow.com/questions/2805102/how-is-pushing-and-popping-defined

Figure 9.4(b)
http://www.mathworks.it/matlabcentral/fx_files/24238/1/queue_line_2.jpg

Even a queue can be implemented using arrays or linked lists but here we will be using linked lists to implement them. Unlike stacks, which require only one pointer to push or pop an element, queue requires two pointers to perform the addition operation and deletion operation respectively. One pointer that points to the starting of the list is called the front.

Value addition: Did you Know

Code for performing enqueue and dequeue operations on a queue

Below is the code for implementing a push operation.

```
void enqueue(int x) //Here "x" is the data input.
{
        if( rear == NULL)                   //if the queue is empty
        {
                rear = new node;            //Asking for space to store a node.
                rear>info = x;              //Assigning value x to the info field.
                rear>ptr = NULL;
        }
        else
        {
                node *q = new node;         //Asking space for a new node
                q->info = x;                // Assigning value of x to the info field.
                q->ptr=NULL;
                rear->ptr = q;              //this line is main diff between stack and queue
                rear = q;           //This node has now been created and rear now points to it.
        }
```

```
}
```

Below is the code for implementing a dequeue operation-:

```
int dequeue()
{
        if( front==rear == NULL)                    //if the queue is empty
        {
                cout<<"Underflow"<<'\n';
                return -1;
          }
        else
        {
                node * q;
                 return (front->info); //we return the value of front to the program
                q=front;
                front=front->ptr;        //we advance front one step.
                delete q;               //we delete the earlier node pointed to by front
         }
}
```

Explanation is avoided for the above code since the comments are self-explanatory.

Source: Self

And the pointer that points to the end node is called rear. In a queue, all nodes are inserted at end only using a rear pointer. On the other hand, deletion of nodes is accomplished by deleting at beginning, same as stack, using the front pointer that points to the starting node. Therefore we will be using two functions here. Those are addatend(int x) and deleteatbeg().These functions, in queue terminology are respectively knows an enqueue and dequeue. Note that reader must use these names only while implementing a queue. And these are the basic functions of a queue. Note that dequeue is different from a deque which means a doubly ended queue.

9.4 Skip lists

Linked list have one serious drawback. They require sequential scanning to locate an element and each node needs to be traversed. Though this can be reduced if the ordered list is sorted however, the worst case still remains O (n^2). Therefore, we require a list that can skip nodes to avoid sequential searching and increase the speed of finding an element in the list. Skip lists comes to rescue here by providing us a way of non-sequential search. A skip list is a data structure for storing a sorted list of items, using a hierarchy of linked lists that connect increasingly sparse subsequences of the items. The efficiency of searching an element in a skip list is as good as that of a binary search and the worst case here equals O (log n) which is less than O (n^2) .

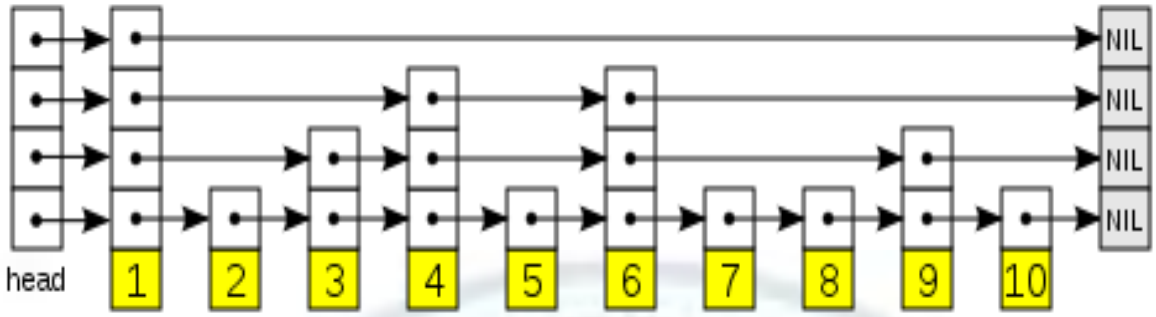


Figure 9.5
http://en.wikipedia.org/wiki/Skip_list

A skip list is built in layers. The bottom layer is an ordinary ordered singly linked list. Each higher layer acts as an "express lane" for the lists below, where an element in layer *i* appears in layer *i*+1 with some fixed probability *p* (two commonly-used values for *p* are 1/2 or 1/4). On average, each element appears in 1/(1-*p*) lists, and the tallest element (usually a special head element at the front of the skip list) in lists.

Value addition: Did you Know
What is a pseudo-code?
Pseudo-code is a compact and informal high-level description of a computer programming algorithm that uses the structural conventions of a programming language, but is intended for human reading rather than machine reading. Pseudo-code typically omits details that are not essential for human understanding of the algorithm, such as variable declarations, system-specific code and subroutines. The programming language is augmented with natural language descriptions of the details, where convenient, or with compact mathematical notation. The benefits of using pseudo-code include human comprehensibility compared to a specific language code. It is commonly used in textbooks and scientific publications that are documenting various algorithms, and also in planning of computer program development, for sketching out the structure of the program before the actual coding takes place. No standard for pseudo-code syntax exists, as a program in pseudo-code is not an executable program. Pseudo-code resembles, but should not be confused with, skeleton programs including dummy code, which can be compiled without errors. Flowcharts can be thought of as a graphical alternative to pseudo-code.
Source: Wikipedia

Value addition: Did you Know

Skip list pseudo code and example

Pseudo code for skip list is given below:

```
P= the non null list on the highest level I;
while e not found and i>=0
        if p->info < e1
                P= a sublist that begins in the predecessor of p on level - - i;
        else if p->info > e1
                if p is the last node on level i
                P = a nonnull sublist that begins in p on the highest level < I;
                i = the number of the new level;
                else p = p->next;
```

A search for a target element e1 begins at the head element in the top list, and proceeds horizontally until the current element is greater than or equal to the target. If the current element is equal to the target, it has been found. If the current element is greater than the target, the procedure is repeated after returning to the previous element and dropping down vertically to the next lower list. The expected number of steps in each linked list is at most 1/*p*, which can be seen by tracing the search path backwards from the target until reaching an element that appears in the next higher list or reaching the beginning of the current list. By choosing different values of *p*, it is possible to trade search costs against storage costs.

Suppose you have to search for element with info as 8 in figure 9.5. Here are the steps-:Following steps will be used for the same:-

1) Start with the top level. Only one element with info 1 is there, so we move to next list.

2) Next list has elements 1, 4 and 6. Again 8 is not here so we move down a level.

3) Next list has element 1, 3, 4, 6 and 9. Here we reach 9 which is greater than 8. So we move back to node previous to 9 (i.e. 6) and move down that node to go below a level where we encounter 7 and then 8. And this is how it works.

Source: Self

Skip list can be implemented by nodes which contain a variable number of link fields for supporting various layers (lists).
Searching in a skip list appears to be efficient; however, the design of skip lists can lead to extremely inefficient insertion and deletion procedures. To insert a new element, first of all the nodes following the nodes have to be restructured, the inserted node has to be added in all the layers and the number of pointers and the values of pointers of almost every node has to be changed. This can be mitigated to some extent by releasing the condition on the position of nodes and keeping restriction only on the number of nodes.

9.5 Self Organizing lists

Alternate to skip lists used to speed up the efficiency of search is self-organizing list, which aims to increase the efficiency of search by dynamically reordering data in a specific way.

The organization depends on the configuration of data. Thus, the stream of data requires reorganizing the nodes already on the list. There are many different ways to organize the list, and this chapter describes the four of them:

1) Move-to-front method- After the desired element is located put it at the beginning of the list. C is desired element in figure below.

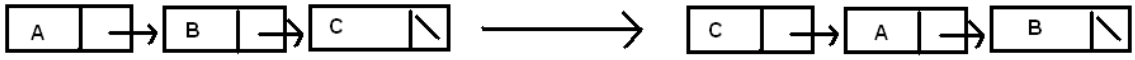


Figure 9.6
Source: Self

2) Transpose method- After the desired element is located swap it with its predecessor unless it is at the head of the list. C is desired element in figure below.

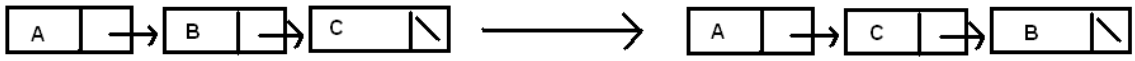


Figure 9.7
Source: Self

3) Count method- Order the list by number of times elements are being accessed. C is desired element in figure below.

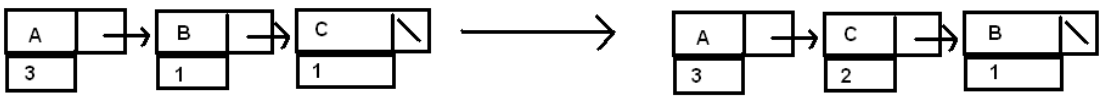


Figure 9.8
Source: Self

4) Ordering method- Order the list using certain criteria natural for the information under scrutiny. C is desired element in figure below.

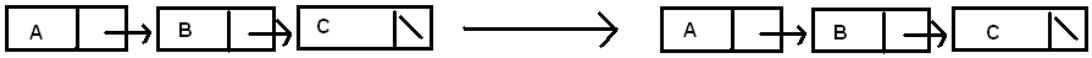


Figure 9.9
Source: Self

Here the desired element may be the element that is being accessed currently. So the list is automatically organized in such a way that next time when that element is accessed, the time required to reach it is less than the previous search. In figure 9.9 above, the list is organized according to ascending values. Hence the list remains the same i.e. A < B < C.

In the first three methods new information searched is stored in a node and that node is added to the end of the list; and subsequent actions as per the method take place. In the fourth method, new information is stored in a node inserted somewhere in the list to maintain the order of the list.

The ordering method uses some properties on basis of which it sorts the information and positions the nodes accordingly. For example, if the nodes contain names of people then list can be organized alphabetically in ascending or descending order. This method is useful when item being searched has a low probability of being on the list, and in that case, scanning the whole linked list is not required. As soon as an element greater than the searched element is found, the search operation is terminated. The count method can become a special case of ordered list if the count of each node is itself considered the information part.

Research has found out that for lists of modest size, the linked list suffices. With the increase in amount of data and in the frequency with which they have to be accessed, more sophisticated methods and data structures need to be used.

Chapter 10: Miscellaneous issues with Linked lists

This chapter deals with different issues/ applications of linked lists in diverse areas like: Applications of linked lists in computer systems, application of linked lists to representing sparse matrices, binary search trees, doubly ended queues and so on.

10.1 Linked representation of sparse matrices

As defined earlier a sparse matrix is a matrix which has a large number of zero/NULL values as compared to other values. There is no empirical formula with the help of which we can decide if some matrix is a sparse matrix or not but it comes from experience whether a matrix is a sparse matrix or not. Sparse matrices have many useful applications like in Numerical analysis- where we need to calculate the inverse of huge sparse matrices of dimension approximately 1000*1000.To represent the matrix in some other form which reduces the space and also gives us lesser values to operate upon.

Figure 10.1
Source: http://ldpcproject.files.wordpress.com/2007/12/matrix.jpg

Figure 10.1 shows a sparse matrix. Carefully notice that most of the elements above in the matrix are zero. In the following paragraph we use linked lists to represent this matrix as described below.

We assume a small matrix and show its representation first and then go on to show the structures/classes which can be used to represent this. Suppose the dimension of the matrix is nXn then we need to make n+n+2 linked lists to represent the matrix i.e. one for each row, one for each column and 2 additional linked lists. If the reader is interested then he/she can also study the array representation form of a sparse matrix which says that for a matrix of mXn dimension if the number of non-zero elements is p then the sparse matrix can be represented by a matrix of dimension (p+1)X3. Space is definitely saved in this case as p << mXn i.e. the total number of elements in the sparse matrix. The three columns are used to store the row number, column number and value of the non-zero element. The extra row i.e. the (p+1)th row is used to store the dimensions of the original matrix, and the total number of non-zero elements. Certainly we can reproduce the sparse matrix by using the alternate form which is logical since there should be absolutely no information loss whatsoever. However we continue with the linked lists representation below. Suppose a small 3X3 matrix shown below.

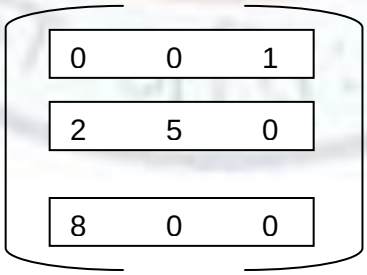


Figure 10.2
Source: Self

Figure 10.2 shows a sparse matrix as the number of non-zero elements is 4 which is less than the total number of zero elements which is five. Now we need to build a matrix shown below by way of coding.

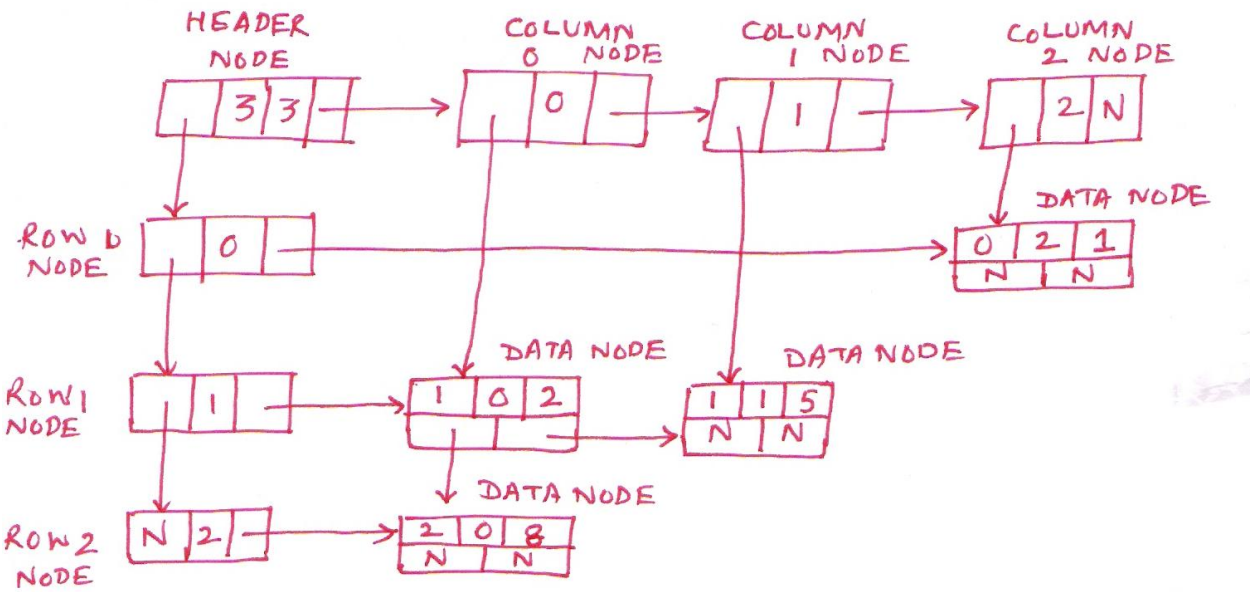


Figure 10.3
Source: self

As we can see the total number of linked lists we have formed is equal to 3+3+2 which is what we said above. If you see the nodes carefully these are just nodes of a doubly linked list which have been joined together in a "matrix" way. Each of the nodes contains two pointer fields, the same as that in a node of a doubly linked list. Of course the information fields are different in each of the node types. We see there are four types of nodes which can be called as a (1) Header Node (2) Column node (3) Row node and (4) Data Node. Description of each type of node is as follows:

(1) Header node: This has two information fields, which contain the original rows and original columns of the original sparse matrix (3 and 3 in the example). Also this has two pointer fields which point to the first row node and first column node.

(2) Column node: There are three column nodes one for each column. It has just one information field which contains the column number starting with 0. There are two pointer fields. The second one points to a column node which is next to the current node. The first pointer field points to the first data node in that column. Had there been no data node then this filed would have contained a NULL. Notice that the last node contains N which is nothing but NULL.

(3) Row node: The row node contains one information field which is the row number and has two pointer fields pointing to the next row node and pointing to the first data node in that row. Notice that the first pointer field of the row node is NULL as there is no row node after it.

(4) Data node: This is slightly more complex and contains three information fields and two pointer fields. The information fields reflect the row number, the column number of the non-zero data and data itself. The pointer fields point to the next data node in the next column and the next data node on the next row, if there is any else they are taken to be NULL.

Value addition: Did you Know

Defined Structures to describe sparse matrix

Programmatically we can use the following structures to describe the different nodes, for representing a sparse matrix.

```
struct cheadnode          //this describes the column node
{
        int colno ;
        struct node *down ;
        cheadnode *next ;
} ;

struct rheadnode          //this describes the row node
{
        int rowno ;
        struct node * right ;
        rheadnode *next ;
} ;

struct spmat              //this describes the special header node
{
        rheadnode *firstrow ;
        cheadnode *firstcol ;
        int noofrows ;
        int noofcols ;
} ;

struct node               //this describes the data node.
{
        int row ;
        int col ;
        int val ;
        node *right ;
        node *down ;
} ;
```

Source: Self

This completes the description of the nodes of a linked list to represent a sparse matrix. The reader can read any book on data structures which gives the implementation of a sparse matrix using linked lists.

10.2 Binary search Tree – (implementation)

Binary search trees were explained briefly in Chapter Eight. This section explains implementation of a binary search tree (or BST) using doubly linked lists. A node of a BST is defined as follows-:

```
class btreenode
{
public:
        btreenode *leftchild ;      // this field will point to the left child
        int data ;                  // this is the information part
        btreenode *rightchild ;     // this field will point to the right child
};
```

Recursive functions are the easiest way to implement a particular operation, as you will soon see. We first give you the code and then the explanation-:

```
  void btree :: buildtree ( int num )
{
        insert ( &root, num ) ;
}

void btree :: insert ( btreenode **sr, int num )
{
        if ( *sr == NULL )
        {
                *sr = new btreenode ;

                ( *sr ) -> leftchild = NULL ;
                ( *sr ) -> data = num ;
                ( *sr ) -> rightchild = NULL ;
                return ;
        }
        else                        // search the node to which new node will be attached
        {
                                    // if new data is less, traverse to left
                if ( num < ( *sr ) -> data )
                        insert ( & ( ( *sr ) -> leftchild ), num ) ;
                else
                                            // else traverse to right
                        insert ( & ( ( *sr ) -> rightchild ), num ) ;
        }
        return ;
}
```

The function void btree :: buildtree() is used to call the insert function and provides **root (*root is of type btreenode and points to the starting location of the tree) and the information value of the node as the argument. Notice that we are passing **root and not just *root. Why are we doing this? This is because if we pass *root as the argument, it will be passed off as "call by value" and a copy of this root will be created and hence a separate copy of tree will created which will be destroyed when the insert function terminates. So, to avoid this we pass a pointer to pointer which points to the root node so changes are reflected back and an "original" tree is created.

In the recursive function insert() , we first check if the root is NULL. If it is NULL, then we create a node, put the info in it as num and then we make both its leftchild and rightchild

point to NULL. If the root already exists, we search the node to which the new node will be attached. Since a node with info value less than that of a current node goes to the left and to the right otherwise, we compare the value of num with every nodes value and move down the tree accordingly till we reach the leafnode(terminating node) and move to its leftchild or rightchild which is decide by the fact whether the value of num is less than or greater than the current node value. Now both leftchild and rightchild of this leaf node will obviously be NULL. So when the function runs its recursive call, the condition of *sr being NULL will be true and then we will be able to insert the node. And this is how it works.
Note that while in BST we first recursively find the position and when the correct position is found, we insert the node.

The iterative code for the same becomes longer in length compared to the recursive counter-part, hence we prefer recursive procedures.

Now we will write a code for traversing the BST in sorted order. This type of traversal is famous by the name of in-order traversal. Here again, the function will be recursive. We present the code below-:

```
void btree :: inorder ( btreenode *sr )
{
        if ( sr != NULL )                       //if the end is not reached
        {
                inorder ( sr -> leftchild ) ;   //run this function again with leftchild a new argument
                cout << "\t" << sr -> data ;    //display value
                inorder ( sr -> rightchild ) ;  // run this function again with leftchild a new argument
        }
}
```

We know that in a BST element to the left of a node is less than elements to the right. In this way, we can clearly make out that leftmost node in the BST contains the smallest value and rightmost node contains the biggest value. So if we have to print element in ascending order, first we will go to the left most node and from there print the value of all nodes in order while moving right horizontally. This is what we achieve using the above function. First it keeps going to the leftmost node till it encounters null and then it starts printing all the values in reverse order in which it came and goes to the right node till the rightmost node printing all the values in ascending order which is exactly what we required. Reader should try to dry run this function on a paper for better understanding.

Value addition: Did you Know
Three different types of traversals
inorder traversal: To traverse a non-empty binary tree in inorder (symmetric), perform the following operations recursively at each node: 1. Traverse the left subtree. 2. Visit the root.

3. Traverse the right subtree.

pre order traversal: To traverse a non-empty binary tree in preorder, perform the following operations recursively at each node, starting with the root node:

1. Visit the root.
2. Traverse the left subtree.
3. Traverse the right subtree.

post order traversal: To traverse a non-empty binary tree in postorder, perform the following operations recursively at each node:

1. Traverse the left subtree.
2. Traverse the right subtree.
3. Visit the root.

Source: Self

Value addition: Did you Know

What is tree-sort?

Given a set of data values if we create a binary search tree using them and perform in-order traversal the values come out in ascending order. This is called a Tree-sort and the worst-case complexity is O (nlogn). This is comparable to the best sorting methods that are available. The big advantage over here is that the data values can be inserted into the binary search tree dynamically hence eliminating the need for a static array. The other sorting techniques require that either the user first specifies the number of elements or we allocate space in the array where they can be stored for example quicksort and mergesort. But we never know in advance as to how many values are going to be sorted, thereby this method is the efficient one. However note that the structure of the tree is not altered, hence we do not have a sorted tree after we perform in-order traversal. So just in case some other application needs to use the sorted values they must be stored in a temporary file which can be written easily or the sorted sequence passed into that application. What about sorting them in the descending order? Well make a binary search tree with the exact opposite rules i.e. the lesser value goes to the right and the greater value goes to the left and perform in-order again.

Source: Self

10.3 Uses of linked lists in operating systems

A linked list is a versatile data structure and finds application in designing of various concepts used in application software and the system software. Here we try to mention some of them. Describing all of them is virtually impossible.

10.3.1 Process queues

As we know that processes are the programs in execution and they must wait for the processor for execution. This time which is called a time slice can be given in several ways. Policies like First come first serve, Shortest time remaining first, Priority queueuing, and round robin policy to name a few. The last one is the most commonly used scheduling policy which is used in operating systems like UNIX, and Windows etc. It's not actually a round robin policy but a variation of certain policies combined together. For example UNIX typically uses a priority based multilevel round robin policy. What it means is that the processes are put at different levels and are normally given time slices/quantum in round robin policy but if a process with high priority comes in then the queue can be over-ridden and the process is given the time quantum first. The processes at various levels then are represented and joined together in the form of circular linked lists. If there are n processes at certain level then according to round robin-policy each gets a time quantum after $(n-1)t_q$ time where t_q is the time quantum as decided by the processor. Whether it be the processes waiting for the CPU, or the processes which are waiting to get into the memory or any set of processes – they are almost always represented by linked lists. Figure 10.4 below shows a typical queue of processes waiting for the processor.

Value addition: Did you Know

What is a process?

In computing, a process is an instance of a computer program that is being executed. It contains the program code and its current activity. Depending on the operating system (OS), a process may be made up of multiple threads of execution that execute instructions concurrently.

A computer program is a passive collection of instructions; a process is the actual execution of those instructions. Several processes may be associated with the same program; for example, opening up several instances of the same program often means more than one process is being executed. Different policies are used for allocation of CPU time to different processes.

Multitasking is a method to allow multiple processes to share processors (CPUs) and other system resources. Each CPU executes a single task at a time. However, multitasking allows each processor to switch between tasks that are being executed without having to wait for each task to finish. Depending on the operating system implementation, switches could be performed when tasks perform input/output operations, when a task indicates that it can be switched, or on hardware interrupts.

A common form of multitasking is time-sharing. Time-sharing is a method to allow fast response for interactive user applications. In time-sharing systems, context switches are performed rapidly. This makes it seem like multiple processes are being executed simultaneously on the same processor. The execution of multiple processes seemingly simultaneously is called concurrency.

For security and reliability reasons most modern operating systems prevent direct communication between independent processes, providing strictly mediated and controlled inter-process communication functionality.

Source: Wikipedia

Figure 10.4 shows some processes which need CPU for execution.

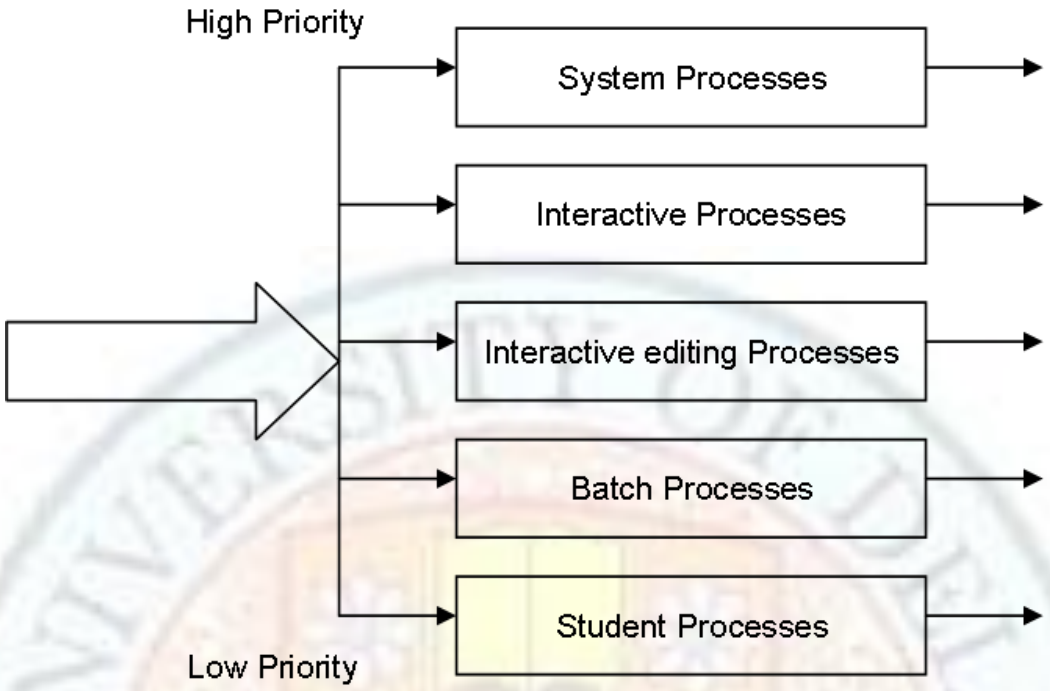


Figure – Multilevel Priority Queue Scheduling

Figure 10.4
Source: http://read.cs.ucla.edu/111/_media/notes/mlq.png?w=&h=&cache=cache

10.3.2 Hashing

A hash table is a data structure that uses a hash function to map identifying values, known as keys, (e.g., a person's name) to their associated values (e.g., their telephone number). It can also be viewed as the mathematical manipulation of a key value to map to a certain location. The hash function is used to transform the key into the index (the hash) of an array element (the slot or *bucket*) where the corresponding value is to be sought. Ideally, the hash function should map each possible key to a unique slot index, but this ideal is rarely achievable in practice (unless the hash keys are fixed; i.e. new entries are never added to the table after creation). Most hash table designs assume that hash collisions—the situation where different keys happen to have the same hash value—are normal occurrences and must be accommodated in some way. In many situations, hash tables turn out to be more efficient than search trees or any other table lookup structure. For this reason, they are widely used in many kinds of computer software, particularly for associative arrays, database indexing, caches, and sets.

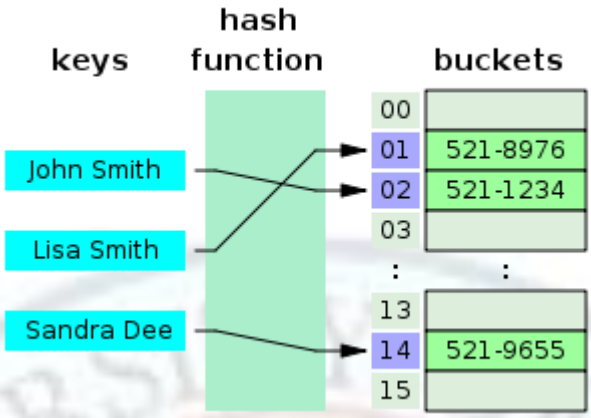


Figure 10.5
http://en.wikipedia.org/wiki/File:Hash_table_3_1_1_0_1_0_0_SP.svg

Hash collisions are practically unavoidable when hashing a random subset of a large set of possible keys. For example, if 2500 keys are hashed into a million buckets, even with a perfectly uniform random distribution, there is a 95% chance of at least two of the keys being hashed to the same slot. Therefore, most hash table implementations have some collision resolution strategy to handle such events. The number of records which are occupied in the hash table divided by the total number of slots is called the load factor.

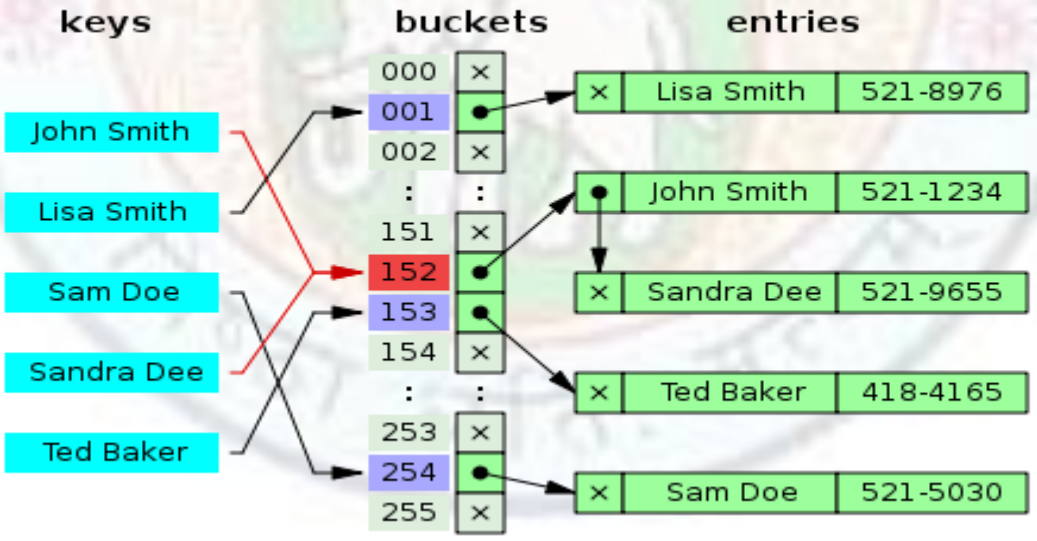


Figure 10.6
http://en.wikipedia.org/wiki/File:Hash_table_5_0_1_1_1_1_1_LL.svg

Figure 10.6 above shows the strategy known as separate chaining, direct chaining, or simply chaining where each slot of the bucket array is a pointer to a linked list that contains

the key-value pairs that hashed to the same location. Lookup requires scanning the list for an entry with the given key. Insertion requires adding a new entry record to either end of the list belonging to the hashed slot. Deletion requires searching the list and removing the element. (The technique is also called open hashing or closed addressing, which should not be confused with 'open addressing' or 'closed hashing'.).

Chained hash tables with linked lists are popular because they require only basic data structures with simple algorithms, and can use simple hash functions that are unsuitable for other methods. The cost of a table operation is that of scanning the entries of the selected bucket for the desired key. If the distribution of keys is sufficiently uniform, the average cost of a lookup depends only on the average number of keys per bucket—that is, on the load factor. Chained hash tables remain effective even when the number of entries *n* is much higher than the number of slots.

Their performance degrades more gracefully (linearly) with the load factor. For example, a chained hash table with 1000 slots and 10,000 stored keys (load factor 10) is five to ten times slower than a 10,000-slot table (load factor 1); but still 1000 times faster than a plain sequential list, and possibly even faster than a balanced search tree.

Value addition: Did you Know

Hashing

Perfect Hashing:

Hash function that is injective—that is, maps each valid input to a different hash value—is said to be perfect. With such a function one can directly locate the desired entry in a hash table, without any additional searching.
Unfortunately, perfect hash functions are effective only in situations where the inputs are fixed and entirely known in advance, such as mapping month names to the integers 0 to 11, or words to the entries of a dictionary. A perfect function for a given set of *n* keys, suitable for use in a hash table, can be found in time proportional to *n*, can be represented in less than 3**n* bits, and can be evaluated in a constant number of operations. There are generators that produce optimized executable code to evaluate a perfect hash for a given input set.
A perfect hash function for a set S is a hash function that maps distinct elements in S to distinct integers, with no collisions. A perfect hash function with values in a limited range can be used for efficient lookup operations, by placing keys from S (or other associated values) in a table indexed by the output of the function.
A perfect hash function for a specific set S that can be evaluated in constant time, and with values in a small range, can be found by a randomized algorithm in a number of operations that is proportional to the size of S. The minimal size of the description of a perfect hash function depends on the range of its function values: The smaller the range, the more space is required. Any perfect hash functions suitable for use with a hash table require at least a number of bits that is

proportional to the size of S.
Using a perfect hash function is best in situations where there is a frequently queried large set, S, which is seldom updated. Efficient solutions to performing updates are known as dynamic perfect hashing, but these methods are relatively complicated to implement. A simple alternative to perfect hashing, which also allows dynamic updates, is cuckoo hashing.

Minimal perfect hash function

A minimal perfect hash function is a perfect hash function that maps n keys to n consecutive integers—usually [0..n−1] or [1..n].

Pearson Hashing

Pearson hashing is a hash function designed for fast execution on processors with 8-bit registers. Given an input consisting of any number of bytes, it produces as output a single byte that is strongly dependent on every byte of the input. Its implementation requires only a few instructions, plus a 256-byte lookup table containing a permutation of the values 0 through 255.
This hash function is a CBC-MAC that uses an 8-bit random block cipher implemented via the permutation table. An 8-bit block cipher has negligible cryptographic security, so the Pearson hash function is not cryptographically strong; but it offers these benefits:

- It is extremely simple.
- It executes quickly on resource-limited processors.
- There is no simple class of inputs for which collisions (identical outputs) are especially likely.
- Given a small, privileged set of inputs (e.g., reserved words for a compiler), the permutation table can be adjusted so that those inputs yield distinct hash values, producing what is called a perfect hash function.

Source: Wikipedia

For separate-chaining, the worst-case scenario is when all entries were inserted into the same bucket, in which case the hash table is ineffective and the cost is that of searching the bucket data structure. If the latter is a linear list, the lookup procedure may have to scan all its entries; so the worst-case cost is proportional to the number *n* of entries in the table. The bucket chains are often implemented as ordered lists, sorted by the key field; this choice approximately halves the average cost of unsuccessful lookups, compared to an unordered list. However, if some keys are much more likely to come up than others, an unordered list with move-to-front heuristic may be more effective which we have seen in the topic on self-organizing lists. More sophisticated data structures, such as balanced search trees, are worth considering only if the load factor is large (about 10 or more), or if the hash distribution is likely to be very non-uniform, or if one must guarantee good performance even in the worst-case. However, using a larger table and/or a better hash function may be even more effective in those cases. Chained hash tables also inherit the disadvantages of linked lists. When storing small keys and values, the space overhead of the next pointer in each entry record can be significant. An additional disadvantage is that traversing a linked list has poor cache performance, making the processor cache ineffective.

10.3.3 Bucket sort

Another example where linked lists are used in a similar fashion is the bucket sort algorithm in which values between 0 and 1 are mapped to a certain number of buckets and if two values map to the same bucket then they are maintained in order by using a sort like insertion sort. Although insertion sort has time complexity $O(n^2)$ when working in n elements but the order comes out to be O (n) as we are not applying insertion sort on the n data values together but actually on the individual elements in a bucket which on an average can be found like total elements/total buckets. The figure below shows 10 data values ranging from 0 to 1 inclusive of these two numbers which are being mapped to 10 buckets. Notice that the values in buckets are sorted in order by using some sorting algorithm like insertion sort and are maintained as linked lists. Hence it makes sense to insert data into buckets by value instead of sorting them after they have been added. The formula used to hash each data value onto a bucket used is taking the ceiling function of the number of buckets multiplied by the data value. Hence value like 0.72 maps to the 8^{th} bucket because the ceiling of 0.72 * 10 = 7.2 is the 8^{th} bucket or bucket number 7 when we are start the numbering from 0.

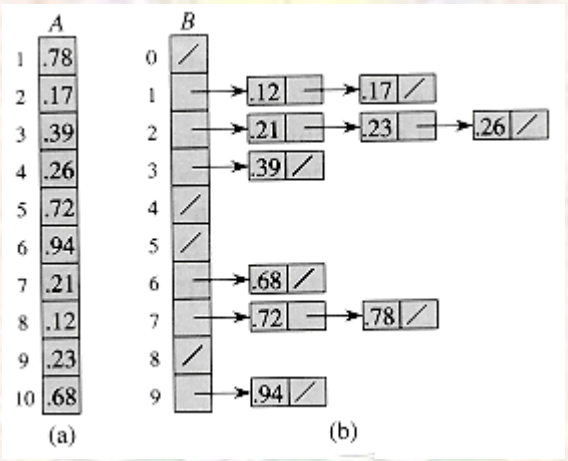


Figure 10.7
http://net.pku.edu.cn/~course/cs101/resource/Intro2Algorithm/book6/181_a.gif

Summary

This chapter introduces the concept of a doubly linked list, followed what are the primary advantages of a doubly linked list by showing the amount of complexity and time it saves by maintaining a "prev" field in each node which point to the node immediate behind.
Then some basic operations were introduced with the doubly linked list and most operations with a single forward traversal are the same as those used with singly linked list. We

harvested the power of a DLL by displaying the contents of it in a reverse order. Then we performed insertion operations and got to know some general tips on how to go about on questions like these.

Concepts like reversing the nodes of a doubly linked list in a single pass, finding the middle node of the doubly linked, circular doubly linked list and it modifications, adding a node, deleting a node in CDLL, application of doubly linked list in the form of a binary tree were discussed.

Discussion on how to diagnose a corrupted linked list with a node having a link field pointing to an illegal node, concepts of stacks and queues as data structures and their various applications, skip lists and its implementation that increases the efficiency of a search operation, how a search algorithm works in a skip list, pros and cons of using a skip list instead on a normal singly linked list, a self organizing lists, various methods that are implemented to reorder a list took place. We found out that while move to front and count method proved inefficient when used on a small file, the outperformed every other method when size of the file is huge. In the end we concluded that for a medium sized file a singly linked list suffices to implement a search operation with decent efficiency.

The sparse matrix can be represented in the linked list form by using nodes of 4 types majorly: special header node, column node, row node and the data node. It saves a lot of space and makes the operations run faster as there are fewer values to operate upon. They can also be represented using the 3 column array format. The binary search tree is just a variation of how nodes are added to a doubly linked list. The fields of the binary search tree are named as info, left and right just to give logical names and good visualization to it. Tree sort is a type of sort which uses a binary search tree as the underlying structure and performs in-order search on it to carry out sorting. Linked lists find several uses inside the computer system software all of which are virtually impossible to describe as there are so many of them. Processes which are in the CPU queue are maintained as circular linked lists or priority queues (implemented again as linked lists). Collision in hashing when resolved with chaining results in formation of linked lists. Bucket sort uses insertion sort on linked lists to sort out elements in the buckets and then combines them together to sort the entire set of data values. Linked lists are amazingly versatile and a thorough understanding of them is imperative for any programmer.

Exercises

7.1 Can you efficiently perform binary search on a DLL? If yes, then write the code assuming the list to be sorted.

7.2 What are the disadvantages of a DLL as compared to a SLL?

7.3 Write a code to display the contents of a DLL in reverse order without making a copy of starting location. Design the code in such a way that the starting pointer points to its initial location after printing.

7.4 Define a linked list that maintains a pointer to the last node too. Then perform the operation of adding the node at the end. Does it make things simpler?

7.5 Write a code for performing sort operation on a DLL.

7.6 Write a function for comparing two DLLs.

7.7 Write a function for efficiently counting the number of nodes in a DLL using both the starting pointer and a tail pointer.

7.8 Implement queue operations using a DLL.

7.9 A deque stands for a doubly end-ed queue in which insertions and deletions can be done at both the ends. Write a program which gives the user a choice to add or delete elements from the both the ends.

7.10 What is the complexity in O –notation for(consider a DLL that has a pointer to the end node as well) -:

a) Adding a node at the end of the DLL.

b) Adding a node at the front of a DLL.

c) Inserting a node before some target node.

d) Inserting a node after some target node.

7.11 Find the error in the following code which adds a node after a given node with value y:

```
void dll::addafter(int y,int x)        //adds value x after given value y
    {
        dnode *t = p;                          //Creating a copy of "p"
        while(t->info != y && t->ptr != NULL)
                t = t->ptr ;

        if(t->ptr == NULL && t->info != y)
                cout<<" Value "<<y<<" does not exist "<<"\n";
      else
        {
                dnode *q = new dnode;
                q->info = x;
                t->next = q;
                q->prev=t;
                q->next = t->next ;
                if(t->next!=NULL)
                t->next->prev=q;

        }
    }
```

8.1 Write a complete program for implementing insertion sort using doubly linked lists.

8.2 Write a destructor for a singly linked list?

8.3 Code a destructor for a doubly linked list. Is it the same as singly linked list?

8.4 Code a destructor for CDLL. Is it the same destructor used for a doubly linked list?

8.5 Write a recursive procedure for counting all the nodes of a circular doubly linked list.

8.6 What is the search complexity of adding/removing a node in a CDLL .

8.7 Write a code for adding a node at the beginning of a CDLL.

8.8 Write a code for adding a node after certain position in a CDLL. The position is given as an argument in the function.

8.9 Draw a binary search tree when following nodes are added in the same order : 5,7,6,2,4,8,10,14,13

8.10 What is the difference between a binary tree and a multi way tree?

8.11 Can you emulate a circular doubly linked list using two pointer variables and a doubly linked list?

9.1 Write a function for finding a loop in a linked list.

9.2 Write a function for finding the node which forms a loop in a given linked list.

9.3 Write a function for calculating the number of nodes in a linked list with an unknown loop.

9.4 Combine all three functions above that diagnose a linked list for a loop, runs a test to detect a loop, runs a process for finding the node which forms the loop, calculates the number of nodes in the loop (optional), and then fixes it according to the user's desire.

9.5 Write a function to implement a push operation in a stack implemented by linked lists.

9.6 Write a function to pop an element out of the stack implemented using a linked list.

9.7 Write a function to enqueue an element into a queue implemented using a linked list.

9.8 Write a function to dequeue an element into a queue implemented using a linked list.

9.9 Why is an enqueue into a queue not performed using a front pointer.

9.10 Why isn't dequeue performed using a rear pointer only? (Hint: Think the restrictions on a queue)

9.11 Can you implement a stack using two queues?

9.12 Can you implement a queue using two stacks?

9.13 Write the steps for searching an element with information 11 in skip list given in 9.5.

9.14 Draw the intermediate steps and show the final queue after the following operations (initially the queue is empty) -:

Enqueue(10);

Enqueue(20);

Dequeue();

Enqueue(13);

Dequeue();

10.1 Write a program to input a sparse matrix from the user and convert it into the linked representation form.

10.2 Write an iterative routine for inserting data values into a binary search tree.

10.3 Try to prove that the complexity of tree-sort in the worst case is O (n logn).

10.4 What is the maximum number of comparisons needed to search a value in a binary search tree?

10.5 Find out what is a process control block and how can a linked list be used in that?

10.6 Assume a singly linked list which contains numbers from 1 to 1000 with one number repeated hence containing a total of 1001 nodes. How do you find out the repeated number in the linked list in a single pass?

10.7 If linked lists can be coded using structures then they can also be coded using unions. Find out what advantage would a union have over structures when used to create a node. Can we make a heterogeneous linked list using this?

10.8 Write a program for carrying out bucket sort. Can bucket sort be applied to values which are not in the range from 0 to 1? (Hint: with some extra cost can we scale down the values in the range of 0 to 1.

Glossary

a-

associative arrays: an abstract data type composed of a collection of unique keys and a collection of values, where each key is associated with one value (or set of values).

b-

balanced search tree: a tree whose height is kept at a balance automatically during insertions and deletions of a large number of nodes.

binary tree: A data structure representing a tree with a restriction that its node can point to at most two nodes.

bucket sort: a sorting algorithm that works by partitioning an array into a number of buckets. Each bucket is then sorted individually, either using a different sorting algorithm, or by recursively applying the bucket sorting algorithm.

c-

cache: a component that improves performance by transparently storing data such that future requests for that data can be served faster. The data that is stored within a cache might be values that have been computed earlier or duplicates of original values that are stored elsewhere. If requested data is contained in the cache (cache hit), this request can be served by simply reading the cache, which is comparably faster. Otherwise search is performed in the main memory.

call by value: Call-by-value evaluation (also referred to as *pass-by-value*) is the most common evaluation strategy, used in languages as different as C and Scheme. If the function or procedure is able to assign values to its parameters, only its local copy is assigned — that is, anything passed into a function call is unchanged in the caller's scope when the function returns.

call by reference: In call-by-reference evaluation (also referred to as *pass-by-reference*), a function receives an implicit reference to the argument, rather than a copy of its value. This typically means that the function can modify the argument- something that will be seen by its caller.

circular doubly linked list: a linked list in which each node points to the next as well as the node previous to it and also the last node points to the first node.

circular linked list: a modification of singly linked list in which the last node points to the first node instead of pointing to NULL.

class: a (keyword)data structure in C++ language that is used for binding data members and member functions into a single entity. Mainly used to implement object oriented programming.

constructor: a term given to a function that is used to initialize the data members of a class. It may be explicitly defined by the user. It is present by default when no explicit constructor is written by the user.

d-

data structure: an entity used to organize data in an acceptable and easy to implement form.

data type: an entity used to create variables that can refer to a specific type of data.

database index: a data structure that improves the speed of data retrieval operations on a database table at the cost of slower writes and increased storage space. Indexes can be

created using one or more columns of a database table, providing the basis for both rapid random look ups and efficient access of ordered records.

delete: In the C++ programming language, the delete operator calls the destructor of the given argument, and returns memory allocated by new back to the heap. A call to delete must be made for every call to new to avoid a memory leak.

dereferencing: A pointer references a location in memory, and obtaining the value at the location a pointer refers to is known as dereferencing the pointer.

destructor: a method which is automatically invoked when the object is destroyed. Its main purpose is to clean up and to free the resources which were acquired by the object along its life cycle and unlink it from other objects or resources invalidating any references in the process. Can be explicitly defined by the user.

doubly linked list: a linked list in which each node points to the next as well as the node previous to it.

dynamic: a term given to events which take place during the execution time of a process instead of taking place in compile time.

h-

hash collision: the situation where different keys happen to have the same hash value and therefore, point to the same slot.
hash function: At the heart of the hash table algorithm is a simple array of items; this is often simply called the hash table. Hash table algorithms calculate an index from the data item's key and use this index to place the data into the array. The implementation of this calculation is the hash function.

i-

insert sort: a simple sorting algorithm, a comparison sort in which the sorted array (or list) is built one entry at a time. It is much less efficient on large lists than more advanced algorithms such as quick sort, heap sort, or merge sort.

inorder traversal: To traverse a non-empty binary tree in inorder (symmetric), perform the following operations recursively at each node:

4. Traverse the left subtree.
5. Visit the root.

6. Traverse the right subtree.

int: data type (keyword) in C language that is used to represent integers.

iteration: term given to methods which use the concept of loops and repeating a process till goal has been reached.

k-

keywords: a term given to words that convey a specific meaning to a programming language.

l-

linear list: a linked list that has no loops, or in other words a linked list which consists of a node which ultimately points to NULL.

link field: a pointer field of a node that contains the address of the node it is supposed to point to.

linked list: a data structure that consists of a sequence of data records such that in each record there is a field that contains a reference (i.e., a *link*) to the next record in the sequence.

m-

main(): the main method of a program that is called by the operating system to execute the program.

multi-way tree: a data structure representing trees that has no restriction on the number of nodes a node may point to.

n-

NULL: a keyword in C++/C programming languages that refers to a pointer that usually points to an invalid memory location that is not available to the user.

new: an operator that allows dynamic memory allocation on the heap. It attempts to allocate enough memory on the heap for the new data and, if successful, returns the address to the newly allocated memory and otherwise throws an exception. It syntax is:

```
p_var = new type(initializer);
```

node: a node is an abstract basic unit used to build linked data structures such as trees, linked lists, and computer-based representations of graphs. Each node contains some data and possibly links to other nodes.

node,child: a node in tree data structure is called a child node with respect to the node which is directly linked with it above.

node,leaf: a node in tree data structure that has no node beneath it. Or in other words a node that has no children or child node.

o-

object: an instance of a class.
ordered list: a list whose elements are sorted by default.

p-

pre order traversal: To traverse a non-empty binary tree in preorder, perform the following operations recursively at each node, starting with the root node:

4. Visit the root.
5. Traverse the left subtree.
6. Traverse the right subtree.

post order traversal: To traverse a non-empty binary tree in postorder, perform the following operations recursively at each node:

4. Traverse the left subtree.
5. Traverse the right subtree.
6. Visit the root.

pointer: a data type that whose value refers directly to (or "points to") another value stored elsewhere in the computer memory using its address. It refers to a location in the memory.
pseudocode:

private: an access specifier (keyword) in C++ programming language that limits the access of data members/methods to the methods present in the enclosing entity only. A class is private by default.

process: another name given to a function/method that will execute at run time.

public: an access specifier that explicitely unrestricts the methods/fields that are in its scope to any outside program or method. Structures are public by default.

q-

queue: A data structure that implements the first-in-first-out(FIFO) implementation. It can be implemented by linked lists/arrays.

r-

recursion: a method of defining functions in which the function being defined is applied within its own definition; specifically it is defining an infinite statement using finite components. In simple words, a recursive function is a function calling itself.

s-

self-referential structure: a data structure that has a pointer within it that is capable of pointing to instances of that structure itself.

singly linked list: a linked list whose nodes contain a single link field that points to the node immediately in front of it.

sparse matrix: a matrix populated primarily with zeros.

stable sorting algorithm: is a sorting algorithm that preserves the order of records with equal keys.

stack: A data structure that implements the first-in-last-out(FILO) implementation. It can be implemented by linked lists/arrays.

struct: a keyword used to declare a structure in C/C++

structure: a (keyword)data structure in C language that is used for binding data members and member functions into a single entity. Traditionally used only for storing fields (data to be operated upon like integer variables etc.) and not the methods.

syntax: a fixed way of writing a command or a statement in a high level language.

t-

traverse: Other name given to scanning a data structure, or accessing each element of a data structure once.

tree: a widely-used data structure that emulates a hierarchical tree structure with a set of linked nodes. Mostly utilizes a doubly linked list for its operations.

Suggested Reading

1. Data Structures and Algorithms in C++, third edition - Adam Drozdek
2. Fundamentals of Data Structures in C++ - Ellis Horowitz, Sartaj Sahni, and Dinesh Mehta
3. Data Structures through C++ - Yashwant Kanetkar
4. Data Structures, Algorithms and Applications in C++, 2nd edition – Sartaj Sahni
5. Data Structures and Algorithm Analysis in C – Mark Allan Weiss