INTERNITY FOUNDATION

PRESENTATION ON LINKED LIST

Presented by:
Lipika Chugh
Rajat Kapoor
Aman Goyal

Arrays Vs Linked Lists

Arrays	Linked list
Fixed size: Resizing is expensive	Dynamic size
Insertions and Deletions are inefficient: Elements are usually shifted	Insertions and Deletions are efficient: No shifting
Random access i.e., efficient indexing	No random access Not suitable for operations requiring accessing elements by index such as sorting
No memory waste if the array is full or almost full; otherwise may result in much memory waste.	Since memory is allocated dynamically(acc. to our need) there is no waste of memory.
Sequential access is faster [Reason: Elements in contiguous memory locations]	Sequential access is slow [Reason: Elements not in contiguous memory locations]

Schematic representation

- Here is a singly-linked list (SLL):

- Each node contains a value(data) and a pointer to the next node in the list

- myList is the header pointer which points at the first node in the list

Insertion at the beginning

There are two steps to be followed:-

a) Make the next pointer of the node point towards the first node of the list
b) Make the start pointer point towards this new node

- If the list is empty simply make the start pointer point towards the new node;

```
void insert_beg(node* new)
{
node* temp;                                  //if the list is empty
            if(head==NULL)
            {   head=new;

                cout<<"\nNode inserted successfully at the beginning";}
                else {
                     temp=head;
                     head=new;
                  } new->next=temp;    //making new node point at
}                                        the first node of the list
```

Inserting at the end

Here we simply need to make the next pointer of the last node point to the new node

```
void insert_end(node* new)
{
node *q=start;
        if(start==NULL)
        {
          start=new;
          cout<<"\nNode inserted successfully at the end…!!!\n";
        }
      else{
                while(q->last!=NULL)
                  q=q->last;
          q->next=new;
          new->next=NULL;
        }
```

Inserting after an element

Here we again need to do 2 steps :-

Make the next pointer of the node to be inserted point to the next node of the node after which you want to insert the node

- Make the next pointer of the node after which the node is to be inserted, point to the node to be inserted

```
void insert_after(int c,node* new)
{
node* q;
q=start;
        for(int i=1;i<c;i++)
       {
          q=q->next;
                 if(q==NULL)
                cout<<"Less than "<<c<<" nodes in the list…!!!";
       }
 new->next=q->next;
 q->next=new;
cout<<"\nNode inserted successfully";
}
```

Deleting a node in SLL

Here also we have three cases:-

- Deleting the first node
- Deleting the last node
- Deleting the intermediate node

Deleting the first node

Here we apply 2 steps:-

- Making the start pointer point towards the 2nd node
- Deleting the first node using delete keyword

```
void del_first()
{
      if(start==NULL)
     cout<<"\nError……List is empty\n";
     else
     {
       node*    temp=start;
       start=temp->link;
       delete temp;
cout<<"\nFirst node deleted successfully….!!!";
     }
}
```

Deleting the last node

Here we apply 2 steps:-

- Making the second last node's next pointer point to NULL
- Deleting the last node via delete keyword

```
void del_last()
{
        if(start==NULL)
            cout<<"\nError….List is empty";
        else{
                    node* q=start;
                            while(q->link->link!=NULL)
                            q=q->link;
                    node* temp=q->link;
                    q->link=NULL;
                    delete temp;
                    cout<<"\nDeleted successfully…";
                }
        }
}
```

Deleting a particular node

Here we make the next pointer of the node previous to the node being deleted ,point to the successor node of the node to be deleted and then delete the node using delete keyword

```
void del(int c)
{
node* q=start;
        for(int i=2;i<c;i++)
        {
          q=q->link;
                if(q==NULL)
                cout<<"\nNode not found\n";
        }
        if(i==c)
        { node* p=q->link;                    //node to be deleted
         q->link=p->link;                 //disconnecting the node p
         delete p;
         cout<<"Deleted Successfully";
        }
}
```

Doubly Linked List

1. **Doubly linked list** is a linked data structure that consists of a set of sequentially linked records called nodes.

2. Each node contains three fields ::

   -: one is data part which contain data only.

   -: two other field is links part that are point or references to the previous or to the next node in the sequence of nodes.

3. The beginning and ending nodes' **previous** and **next** links, respectively, point to some kind of terminator, typically a sentinel node or null to facilitate traversal of the list.

NODE

A doubly linked list contain three fields: an integer value, the link to the next node, and the link to the previous node.

DLL's compared to SLL's

- Advantages:
  - Can be traversed in either direction (may be essential for some programs)
  - Some operations, such as deletion and inserting before a node, become easier
- Disadvantages:
  - Requires more space
  - List manipulations are slower (because more links must be changed)
  - Greater chance of having bugs (because more links must be manipulated)

Inserting at beginning

```
void insert_beg(node *p)
{
  if(start==NULL)
  {
    start=p;
    cout<<"\nNode inserted successfully at the beginning\m";
  }
  else
  {
  node* temp=start;
  start=p;
  temp->previous=p;
  p->next=temp;
  cout<<"\nNode inserted successfully at the beginning\n";
  }
```

Inserting at the end

```
void insert_end(node* p)
{
      if(start==NULL)
      {
      start=p;
      cout<<"\nNode inserted successfully at the end";
      }
      else
      {
       node* temp=start;
       while(temp->next!=NULL)
         {
         temp=temp->next;
         }
       temp->next=p;
       p->previous=temp;
       cout<<"\nNode inserted successfully at the end\n";
      }
}
```

Inserting after a node

Making next and previous pointer of the node to be inserted point accordingly

Adjusting the next and previous pointers of the nodes b/w which the new node accordingly

```
void insert_after(int c,node* p)
{
    temp=start;
    for(int i=1;i<c-1;i++)
    {
    temp=temp->next;
    }
    p->next=temp->next;
    temp->next->previous=p;
    temp->next=p;
    p->previous=temp;
    cout<<"\nInserted successfully";
}
```

Deleting a node

- Node deletion from a DLL involves changing *two* links
- In this example,we will delete node b

- We don't have to do anything about the links in node b
- Garbage collection will take care of deleted nodes
- Deletion of the first node or the last node is a special case

```
void del_at(int c)
{
    node*s=start;
      {
     for(int i=1;i<c-1;i++)
     {
      s=s->next;
     }
     node* p=s->next;
     s->next=p->next;
     p->next->previous=s;
     delete p;
     cout<<"\nNode number "<<c<<" deleted successfully";
      }
   }
}
```

APPLICATIONS OF LINKED LIST

1. Undo functionality in Photoshop or Word (a linked list of state)
2. Image viewer – Previous and next images are linked, hence can be accessed by next and previous button.
3. Previous and next page in web browser – We can access previous and next url searched in web browser by pressing back and next button since, they are linked as linked list.
4. Music Player – Songs in music player are linked to previous and next song. you can play songs either from starting or ending of the list.

THANK YOU