Singly Linked List

Doubly Linked List (Basic)

{Insertion}

Insert At Beginning ()

```
struct Node *node=(struct Node *) malloc (sizeof(struct Node));
node->next=node->prev=NULL;
node->data=data;
if(start==NULL)
{
start=node;
end=node;
}
else
{
struct Node *temp;
temp=start;
temp->prev=node;
start=node;
node->next=temp;
}
size++;
```

Insert At End ()

```
struct Node *node=(struct Node *) malloc (sizeof(struct Node));
node->next=node->prev=NULL;
node->data=data;
if(start==NULL)
{
start=node;
end=node;
}
else
{
```

Singly Linked List

```
struct Node *temp=end;
node->prev=temp;
temp->next=node;
end=node;
}
size++;
```

Insert At Position ()

```
struct Node *node=(struct Node *) malloc (sizeof(struct Node));
node->next=node->prev=NULL;
node->data=data;
int position;
printf("Enter position: ");
scanf("%d",&position);
if(position<0) position=0;
if(position>=size) position=size;
if(position==0)
{
if(start==NULL)
{
start=end=node;
}
else
{
node->next=start;
start->prev=node;
start=node;
}
}
else
{
int i=0;
struct Node *temp=start;
```

Singly Linked List

```
struct Node *p;
while(i<position)
{
p=temp;
temp=temp->next;
i++;
}
p->next=node;
node->prev=p;
node->next=temp;
temp->prev=node;
}
size++;
```

{Deletion}

deleteAtBegining ()

```
struct Node *temp=start;
if(size==1)
{
start=end=NULL;
free(temp);
}
else
{
struct Node *p=temp->next;
start=p;
p->prev=NULL;
free(temp);
}
size--;
```

Singly Linked List

deleteAtEnd ()

```
struct Node *temp=end;
if(size==1)
{
start=end=NULL;
free(temp);
}
else
{
struct Node *p=temp->prev;
p->next=NULL;
end=p;
free(temp);
}
size--;
```

deleteAtPosition()

```
struct Node *temp;
int position;
printf("Enter position: ");
scanf("%d",&position);
if(position<0) position=0;
if(position>=size) position=size;
if(position==0)
{
if(size==1)
{
temp=start;
start=end=NULL;
free(temp);
}
else
```

Singly Linked List

```
{
struct Node *p;
temp=start;
p=temp->next;
start=p;
p->prev=NULL;
free(temp);
}
}
else
{
int i=0;
struct Node *p;
temp=start;
while(i<position)
{
p=temp;
temp=temp->next;
i++;
}
printf("Temp: %d\n",temp->data);
p->next=temp->next;
end=p;
free(temp);
}
size--;
```

Doubly Linked List (OOPS + Template)

Functionality Code

Singly Linked List

SinglyLinkedListNode :-

```
template<class T>
class DoublyLinkedListNode
{
private:
DoublyLinkedListNode *prev;
DoublyLinkedListNode *next;
T data;
DoublyLinkedListNode(T);
friend class DoublyLinkedList<T>;
friend class DoublyLinkedListIterator<T>;
};
tempate<class T>
DoublyLinkedListNode<T>::DoublyLinkedListNode(T data)
{
this->T=T;
this->prev=NULL;
this->next=NULL;
}
```

SinglyLinkedListIterator :-

```
template<class T>
class DoublyLinkedListIterator
{
private:
DoublyLinkedList<T> *node;
DoublyLinkedListIterator(DoublyLinkedListIterator<T> *);
public:
boolean hasNext();
T next();
boolean hasPrev();
T prev();
friend class DoublyLinkedList<T>;
```

Singly Linked List

```
};
template<class T>
DoublyLinkedListIterator<T>::DoublyLinkedListIterator(DoublyLinkedListIterator<T> *node)
{
this->node=node;
}
template<class T>
boolean DoublyLinkedListIterator<T>::hasNext()
{
return this->node!=NULL;
}
template<class T>
T DoublyLinkedListIterator<T>::next()
{
if(this->node==NULL) return 0;
T data;
data=this->node->data;
this->node=this->node->next;
return data;
}
template<class T>
boolean DoublyLinkedListIterator<T>::hasPrev()
{
return this->node!=NULL;
}
T DoublyLinkedListIterator<T>::prev()
{
if(this->node==NULL) return 0;
T data;
data=this->node->data;
this->node=this->node->prev;
return data;
}
```

Singly Linked List

SinglyLinkedList :-

```
template<class T>
class DoublyLinkedList
{
private:
DoublyLinkedListNode<T> *start;
DoublyLinkedListNode<T> *end;
int size;
public:
DoublyLinkedList();
DoublyLinkedList(const DoublyLinkedList<T> &);
virtual ~DoublyLinkedList();
DoublyLinkedList<T>& operator=(DoublyLinkedList<T>);

void add(T);
void insert(int,T);
T remove(int);
void clear();
T get(int);
int getSize();
DoublyLinkedListIterator * getIterator();
};
template<class T>
DoublyLinkedList<T>::DoublyLinkedList()
{
this->start=NULL;
this->end=NULL;
this->size=0;
}
template<class T>
DoublyLinkedList<T>::DoublyLinkedList(const DoublyLinkedList<T> &otherDoublyLinkedList)
{
```

Singly Linked List

```
this->start=NULL;
this->end=NULL;
this->size=0;
DoublyLinkedList<T> *node;
node=otherDoublyLinkedList.start;
while(node!=NULL)
{
this->add(this->data);
node=node->next;
}
}
template<class T>
virtual DoublyLinkedList<T>::~DoublyLinkedList()
{
this->clear();
}
template<class T>
DoublyLinkedList<T> & DoublyLinkedList<T>::operator=(DoublyLinkedList<T> otherDoublyLinkedList)
{
this->clear();
DoublyLinkedList<T> *node;
node=DoublyLinkedList.start;
while(node!=NULL)
{
this->add(this->data);
node=node->next;
}
return * this;
}
template<class T>
void DoublyLinkedList<T>::add(T data)
{
DoublyLinkedList<T> *node;
```

Singly Linked List

```
node=new DoublyLinkedList<T>(data);
if(this->start==NULL)
{
this->start=node;
this->end=node;
}
else
{
end->next=node;
this->node->prev=end;
end=this->node;
}
this->size++;
}
template<class T>
void DoublyLinkedList<T>::insert(int position,T data)
{
if(position<0) position=0;
if(position>this->size) position=this->size;
if(position==this->size)
{
this->add(data);
return;
}
DoublyLinkedListNode<T> *node;
node=new DoublyLinkedList<T>(data);
if(position==0)
{
if(this->start==NULL)
{
this->start=node;
this->end=node;
this->size++;
```

```
return;
}
else
{
this->start->prev=node;
node->next=start;
this->start=node;
this->size++;
return;
}
}
else
{
DoublyLinkedListNode<T> *p,*temp;
temp=this->start;
int i=0;
while(i<position)
{
p=temp;
temp=temp->next;
i++;
}
p->next=node;
node->prev=p;
temp->prev=node;
node->next=temp;
this->size++;
}
}
template<class T>
T DoublyLinkedList<T>::remove(int position)
{
if(position<0 || position>=this->size) return 0;
```

Singly Linked List

```
T data;
DoublyLinkedListNode<T> *temp;
if(this->size==1)
{
temp=this->start;
data=temp->data;
this->start=NULL;
delete temp;
this->size=0;
return data;
}
else
{
temp=this->start;
DoublyLinkedList<T> *p;
int i=0;
while(i<position)
{
p=temp;
temp=temp->next;
i++;
}
data=temp->data;
p->next=temp->next;
temp->next->prev=p;
delete temp;
this->size--;
return data;
}
}
template<class T>
void DoublyLinkedList<T>::clear()
{
```

```
while(this->size>0) this->remove(0);
}
template<class T>
T DoublyLinkedList<T>::get(int)
{
}
template<class T>
int DoublyLinkedList<T>::getSize()
{
return this->size;
}
template<class T>
DoublyLinkedListIterator * DoublyLinkedList<T>::getIterator()
{
return new DoublyLinkedListIterator<T>(this->start);
}
```