

## Session 03

# Vectors and Linked Lists

# Overview

---

- **Linked List(LL)**
  - Definition
  - Comparison with Arrays
  - Implementation from scratch using java
  - `java.util.LinkedList`
- **Variants of Linked List**

# Overview (Cont...)

---

- Vector
  - Definition and Operations
  - Implementation from scratch using java
  - `java.util.Vector`
- Stacks and Queues using LL
- Application of Linked list
  - Sparse table

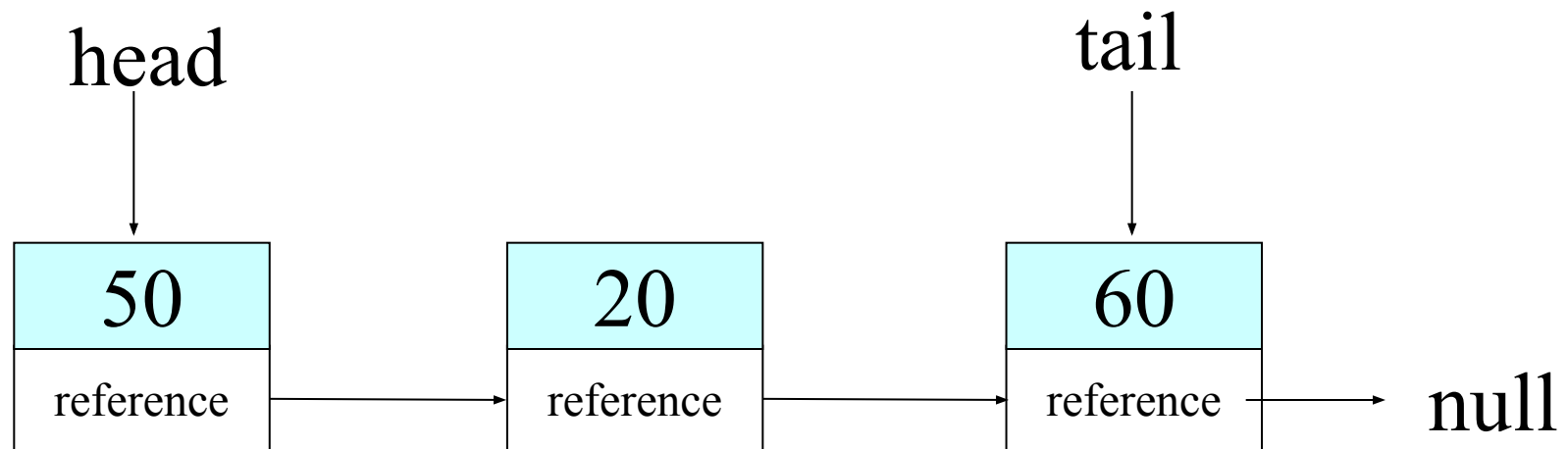
# Linked Lists

---

- Sequence of elements strung together.
- Each element can have at the most one successor and one predecessor.
- Each keeps a reference of its successor.
- Also called singly linked list.
- Insertion and deletion can be at arbitrary positions.
- More general than stack and queue.

# Linked Lists

---



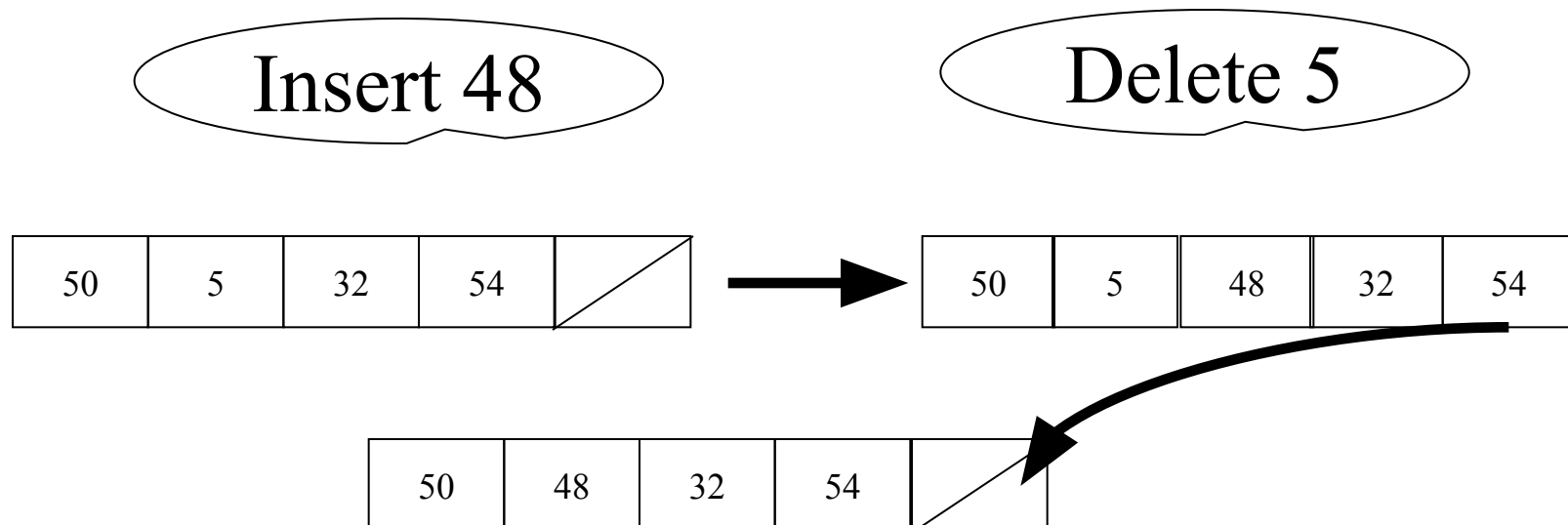
# Operations

---

- Initialization - creation of empty list
- Insert
  - insertBeginning(), insertBefore(), insertAfter(), insertEnd()
- Delete
  - deleteBeginning(), deleteBefore(), deleteAfter(), deleteEnd()
- Traversal/Searching

# Array As a List

- Simple implementation by rearranging array elements
  - insert - move all subsequent elements down
  - delete - move all subsequent elements up
- Too Expensive



# Lists Using Array

---

- Using an array of Nodes
  - Each node holds index to the next node in the list.

```
class Node{
    private int item;
    private int next;
    public Node(int it)
    {item = it;next = -1;}
    public void setNext(int n)
    { next = n; }
    public void setItem(int it)
    { item = it; }
}
```



# List using Array

```
Node[] SLL =  
new Node[4];
```

item  
next

108	134	205	76
3	-1	0	1

Tail

Head

# Lists Using References

---

Each node is dynamically allocated as required.

```
public class Node{  
    private int item; private Node next;  
    public Node(){this(-1,null);}   
    public Node(int item, Node next)  
    {this.item = item; this.next = next;}  
    public void setNext(Node next)  
    {this.next=next;}  
    public Node getNext(){return next;}  
    public int getItem() {return item;}
```

# Lists Using References

---

```
public class LinkedList{
    Node head; // First element in LL
    Node tail; // Last element in LL
    // Initialization of LL
    public void LinkedList(){head = tail = null;}
    public boolean isEmpty(){return head == null;}
    public void insertBeginning(int item){// slide}
    public void insertEnd(int item){//Exercise}
    public void insertAfter(Node ref,int item){//Exercise}
    public void insertBefore(Node ref, int
        item){//Exercise}
```

# Lists Using References

---

```
public void delBeginning() { //Exercise}
public void delete(node) { // Exercise}
public void delEnd() { // special case of delete, Exercise}
public void delAfter(Node) { // Exercise}
public void delBefore(Node) { // Exercise}
public void traverse() { // Exercise}
}
```

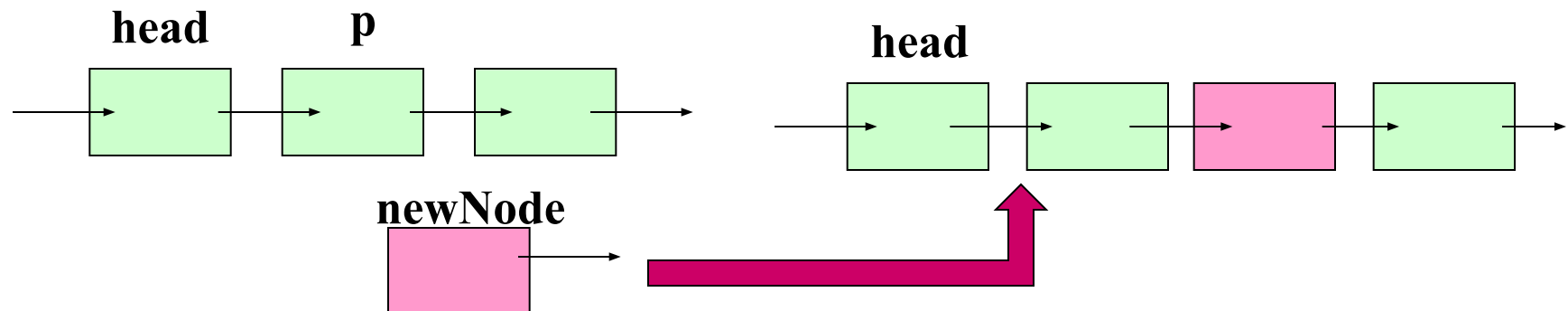
# Insertion At The Beginning

---

```
Public void insertBeginning(int item) {  
    Node temp = new Node(item); // create new node  
    temp.setNext(head); // make next of new node to head  
    head = temp;           // reset head to new node  
    if(tail == null)  
        tail = head;      // new node is the first node  
}
```

# Insertion at Middle

```
public void insertAfter(Node p, int item){  
    Node newNode = new Node(item); // create new node  
    newNode.setNext(p.getNext()); // make next of new  
                                   node to next of p  
    p.setNext(newNode); // set p's next to new node  
    if(tail == p) tail = newNode; // p was the last node  
}
```



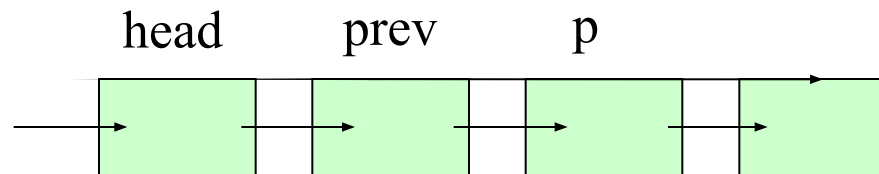
# Insertion Before / Delete

---

- These operations require the reference of previous element, which is not accessible from the current element.
- Has to travel from head, keeping track of previous element.

# delete(Node p)

```
{if (isEmpty()){ // Error; return }
  Node prev = null;
  for(Node cur=head; cur!=p; cur=cur.getNext())
    {prev = cur;}
  if(prev == null) // p == head
    head = head.getNext();
  else{
    prev.setNext(cur.getNext());
  }
  if(tail == cur) tail = prev;
}
```





# Lists and Arrays

---

- Both represent a sequence.
- Contiguity indicates next element in array; explicit indication in list.
- Therefore easy to change next element in a list.
- Insertion: only few references need to be changed; an array requires moving many elements.
- Random access of elements not possible unlike array.
- Normally storage is not pre-allocated in list; but this is not the critical issue.

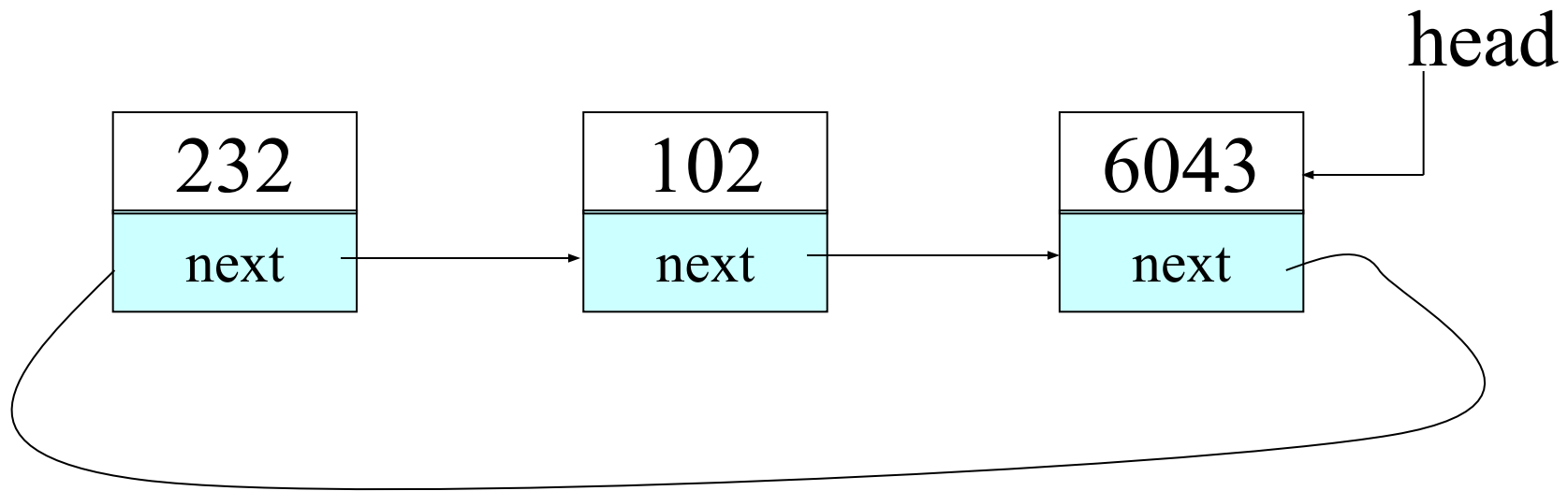
# Other List Types

---

- Circular Linked List (CLL)
- Doubly Linked List (DLL)
- Circular Doubly Linked List (CDLL)

# Circular Linked List

- First node is made the successor of the last node i.e. last node's next reference is to the first node in the list.
- Usage - round robin scheduling of processes on CPU.



# Circular Linked List

---

- Operations
  - Initialize – create empty CLL
  - isEmpty()
  - Insertion/Deletion at beginning/end/middle of the list
  - Traversal
- Implementation
  - `LinkedList` class can be modified, and requires **only** `head` reference to hold the circular linked list.
  - `head` refers to last node in the Circular Linked List.

# Circular Linked List

---

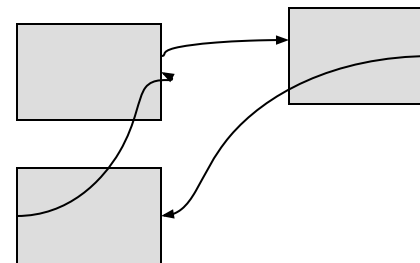
- Insertion at the Beginning

```
Node newNode = new Node(item);  
if(head == null){  
    head=newNode; // CLL was empty  
}  
else newNode.setNext(head.getNext());  
head.setNext(newNode);
```

# Circular Linked List

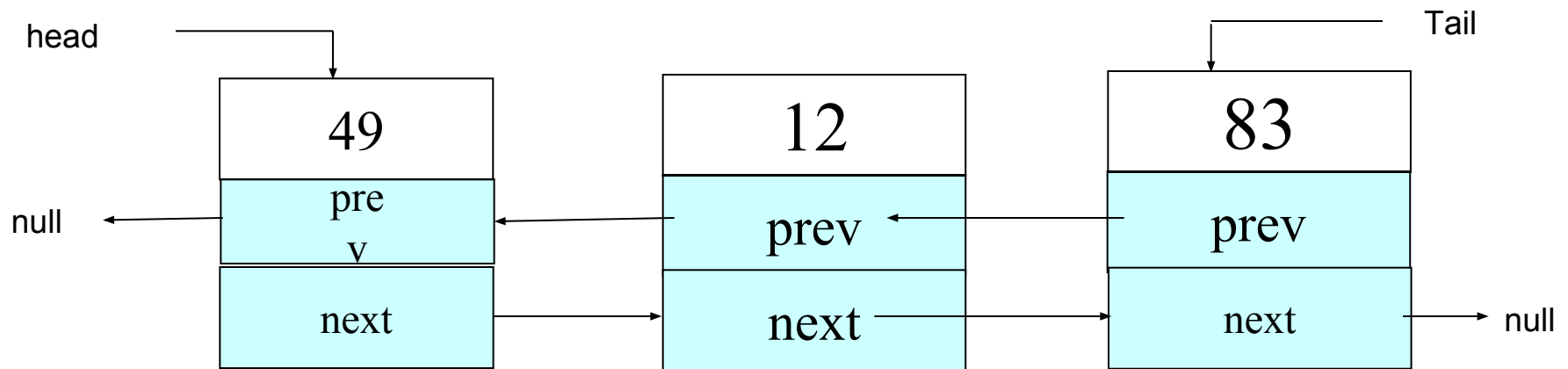
---

- Insertion before/delete
  - Issues are same as those in Linked List



# Doubly Linked List

- Moving up in the list from a given node was difficult.
- We keep reference to predecessor node and successor node, in every node.
- More space requirements.
- Updates now become more complex.



# Doubly Linked List

---

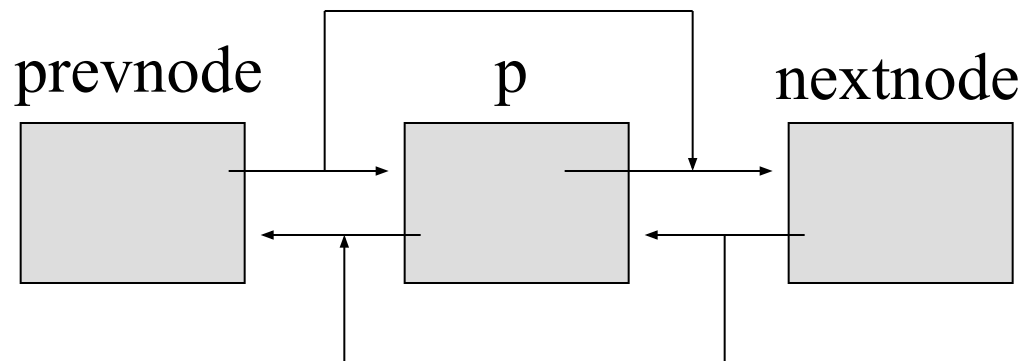
- Operations
  - Initialize – create empty DLL
  - isEmpty()
  - Insertion/Deletion beginning/end/middle of the list
  - Traversal
- Implementation
  - `class Node` should be modified to hold reference to previous element also.
  - Insertion and deletion methods of `LinkedList Class` need to take care of updating these references.



# Deletion in DLL

- **No need to traverse the list to locate the previous element.**

```
// p's previous node refer to p's next node  
p.getPrev().setNext(p.getNext());  
// p's next node refer to p's previous node  
p.getNext().setPrev(p.getPrev());  
// Special care for DLL with single node
```



# Vectors

---

- Array without size constraints.
- Elements can be accessed randomly using index.
- Operation
  - Insertion (at a particular index)
  - Deletion (at a particular index)
  - Length (Number of elements)
  - Traversal

# Vectors – Implementation Using Array

---

```
Class Vector{
    private int length; // Total elements so far
    private int size; // Current space available
    private Object[] arr; // Array holding elements
    public Vector() {
        size = 10; length = 0;
        arr = new Object[size];
    }
}
```

# Vectors – Implementation Using Array

---

```
public void insertAt(int index, Object item);  
// Insertion at 'index'; shifts elements down  
public int deleteAt(int index);  
// Deletion at 'index'; shifts elements up  
public void append(Object item);  
// Append at the end  
public void change(int index, Object item);  
// Modify the element at the 'index'  
public int numElements() { return length; }
```

# Vectors – Implementation Using Array

---

```
// Total number of elements  
public void traversal() {  
    for(int ii=0; ii<size; ii++)  
        if(arr[ii] != null)  
            System.out.println(arr[ii]);  
}
```

# Vectors – Implementation Using Array

---

```
public void insertAt(int index, Object item){
    if(index >= size ) {
        int oldSize = size ;
        size = index  + 10;
        tmpArr = new Object[size];
        for(int ii=0; ii<oldSize;ii++)
            tmpArr[ii] = arr[ii];
        arr = tmpArr;
    }
    arr[index]=item; length++;
}
```

# Vectors – Implementation Using Array

---

```
else if( length + 1 == size ){
    int oldSize = size ;    size + = 10;
    tmpArr = new Object[size];
    for(int ii=0; ii<index;ii++)
        tmpArr[ii] = arr[ii];
    tmpArr[index] = item;
    for(int ii=index+1; ii ≤ oldSize;ii++)
        tmpArr[ii] = arr[ii-1];
    arr = tmpArr; length++;
}
```

# Vectors – java.util.Vector

---

- Generalize to store any object
  - protected Object elementData[] // stores data
  - int elementCount // number of elements
- Insertion at particular index
  - void insertElementAt(Object elem, int index)
- Deletion at particular index
  - void removeElementAt(Object elem, int index)
- Length
  - int capacity()
- Traversal
- Supports many more methods



# Vectors – java.util.Vector

---

## Traversal

```
Vector v = new Vector(10); // Vector with size 10
Enumeration enum = v.elements();
int item;
while (enum.hasMoreElements()) {
    item = ((Integer) enum.nextElement()).intValue();
    System.out.println(item);
}
```

# Stack using LL

---

Wrap the operation of `LinkedList` class in the operations of `Stack` class

```
class Stack{
    Public stack(){ list = new list(); }
    Public boolean isEmpty(){ return list.isEmpty(); }
    Public void push(int item){
        list.addBeginning(item); }
    Public int pop(){ return list.delBeginning(); }
    Public int topEl(){ int item = pop(); push(item);
                        return item;}
    Private LinkedList list; // data member of class
                             stack;
}
```

# Queue using LL

---

Wrap the operation of `LinkedList` class in the operations of `Queue` class

```
class Queue{
    Public queue() { list = new list(); }
    Public boolean isEmpty(){
        return list.isEmpty();}
    Public void insert(int item){
        list.addBeginning(item);}
    Public int remove(){ return list.delEnd(); }
    Private LinkedList list; // data member of
                             class Queue
}
```

# Applications

---

- Adding two big integers
  - 1231312313121312321312 +  
131231231231231
- Library management
- Sparse Table

# Sparse Table: Problem Statement... Facts

---

- A university has 10, 000 students & 500 courses
- One student can take  $\leq 5$  courses per year
- One course can have  $\leq 200$  students
- Test grades for each course are – A+, A, B+, B, C+, C, D+, D, F

# Sparse Table: Problem Statement... Requirements

---

- List students taking a course
- List courses taken by a student
- Grades obtained by a student
- Average grade per course

# Sparse Table: Problem Statement...Constraints

---

- Time complexity
- space complexity

# Sparse Table: Approaches

---

1. 2D array of students vs. courses with each cell storing grade of the student for the course
2. Two 2D arrays
  1. One array stores all the courses taken by a student; a cell represents a course and grade.
  2. Second array stores all the students taking a particular course and cell contains student and grade.
3. Linked list of students and courses



# Sparse Table .. Approach 1

		Student				
		1	2	.....		10000
C o u r s e	1	a				
	2		i	d		c
	.		c	f		
	.			b		
	500	f				a

# Sparse Table .. Approach 1

---

- Space required
  - 10000 students X 500 classes X 1 byte grade = 5MByte space
- Time complexity for the operations
  - List of students taking the course ... linear
  - List of courses taken by the student ... linear

# Sparse Table .. Approach 2

- Each column contains the list of courses taken by a student.
- Each cell contains the course id and grade by the student in course.

	1	2	.....	10000
1	12, a			
2		33, i	22, d	490, c
3		254, c	333, f	
4			175, b	
5	140, f			435, a

STD-CRS Table

# Sparse Table .. Approach 2

- Each column contains the list of students taking the course.
- Each cell contains the student id and grade by the student in the course.

	1	2	.....	500
1	120, a			
2		330, i	220, d	4900, c
.		2540, c	3330, f	
.			1750, b	
200	1400, f			4350, a

CRS-STD Table

# Sparse Table .. Approach 2

- 
- Space required (assume 3 bytes/ cell)
    - $\text{Space(STD-CRS)} + \text{Space(CRS-STD)} = 450\text{KB}$
  - Time complexity for the operations
    - List of students taking the course ... from CRS-STD
    - List of courses taken by the student ... from STD-CRS
  - If one course allows say 500 students, every column must be of size 500.

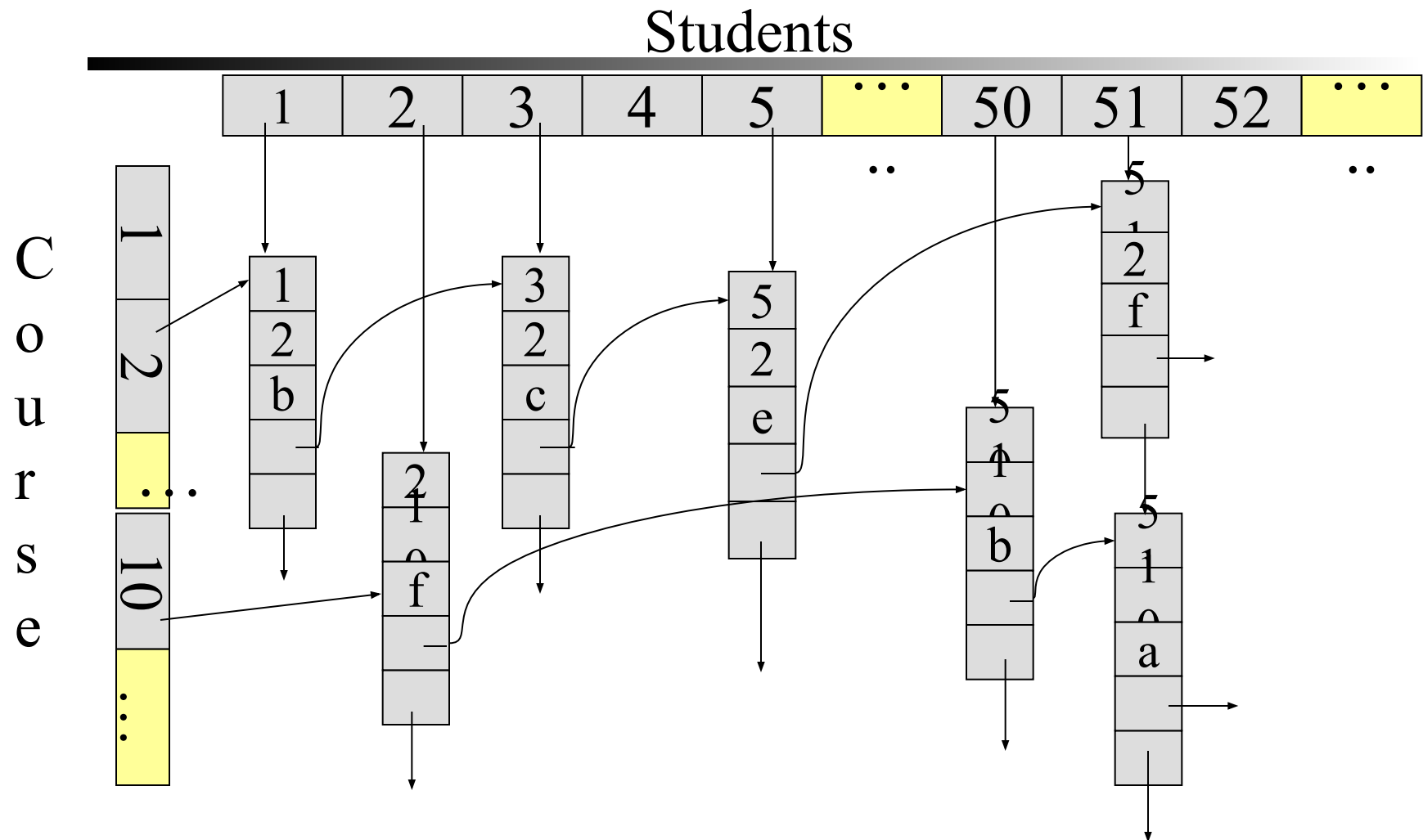
# Sparse Table .. Approach 3

---

- Linked List of courses
- Linked List of students

```
Class StdCrsNode{  
    int classId, stdId;  
    byte grade;  
    StdCrsNode nextCrs, nextStd;  
}
```

# Sparse Table .. Approach 3



# Sparse Table .. Approach 3

---

- Space complexity
  - Size of nodes in both the Linked list  $\approx$  121KB
- Time Complexity
  - List of students taking the course
    - follow `nextStd` reference
    - Constant time (  $\leq 200$  )
  - List of courses taken by the student ... from student list
    - follow `nextCrs` reference
    - Constant time (  $\leq 5$  )



# Summary

---

- SLL, Operations on SLL, and its variants.
- Differences between array and LL.
- Implementation of stacks and queues using LL.
- Vector class and operations on it.