Linked Lists

Overview

- Linked List(LL)
  - Definition
  - Comparison with Arrays
  - Implementation from scratch using C++
- Variants of Linked List

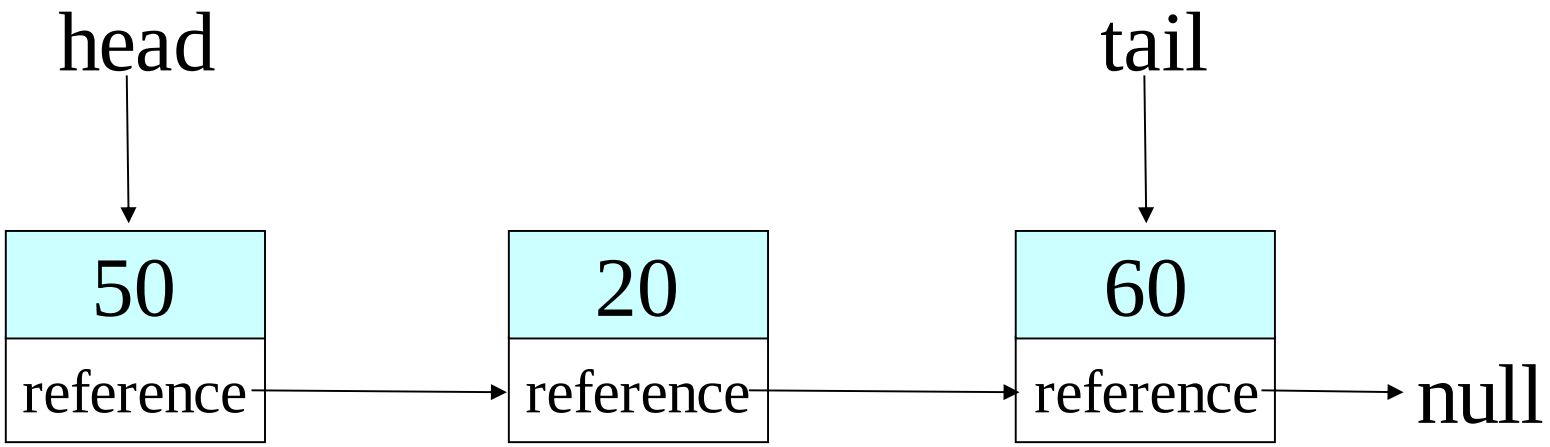
Overview (Cont…)

- 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

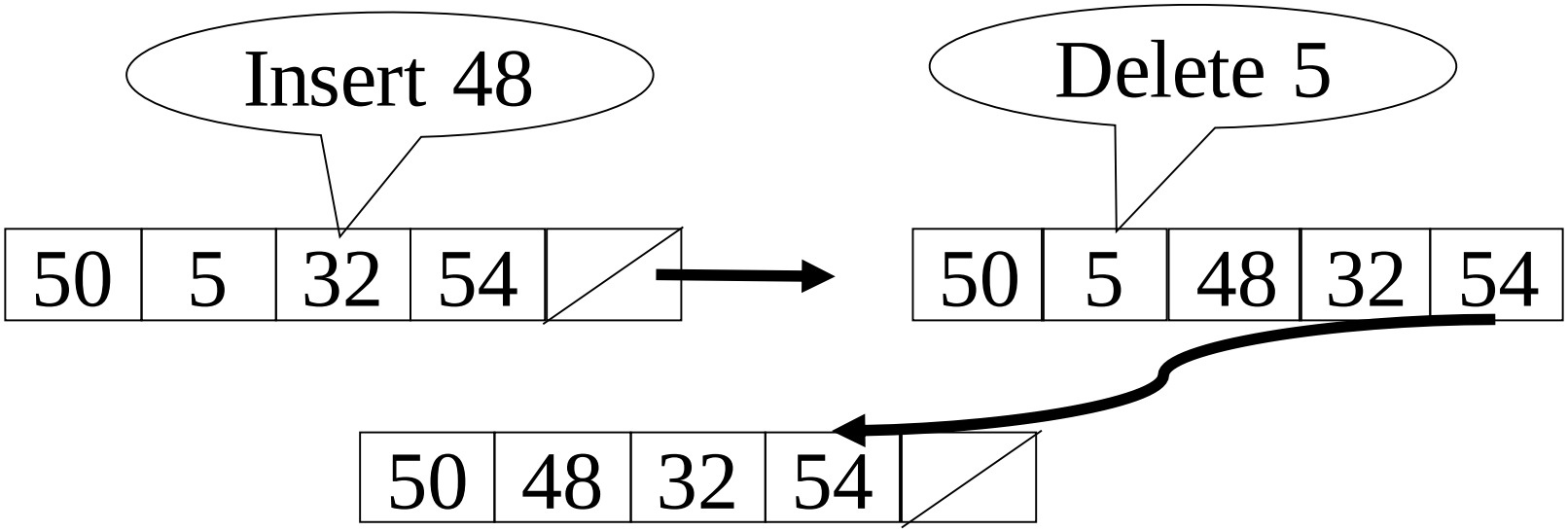


Operations

- Initialization - creation of empty list
- Insert
  - insert_beginning(), insert_before(), insert_after(), insert_end()
- Delete
  - delete_beginning(), delete_before(), delete_after(), delete_end()
- 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



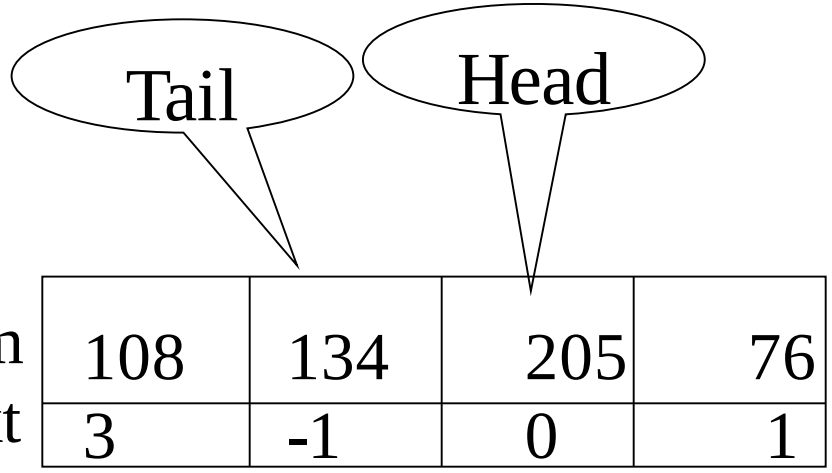
Lists Using Array

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

```
class Node{
  private:
        int item;
        int next;
  public:
        Node(int it){item = it;next = -1;}
          void set_next(int n){ next = n;}
        void set_item(int it){item = it;}
};
```

List using Array

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



Lists Using References

- Each node is dynamically allocated as required.

```
typedef struct Node{
        int item;
        Node *next;
};
```

Lists Using References

```
class LinkedList{
    private:
            Node *head; //First element in LL
 //Initialization of LL
 public:
       LinkedList(){head = NULL;}
       bool is_empty(){return (head == NULL);}
       void insert_beginning(int item){// slide}
       void insert_after(Node ref,int item){//Exercise}
      void insert_before(Node ref, int item){//Exercise}
```

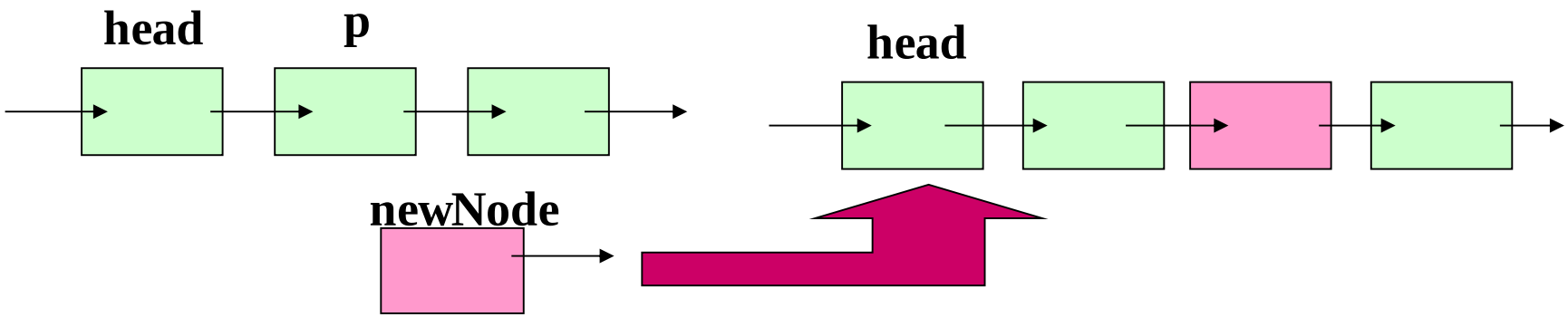
Lists Using References

```
 void del_beginning(){//Exercise}
 void del_after(Node){//Exercise}
 void del_before(Node){//Exercise}
 void traverse() {//Exercise}
};
```

Insertion At The Beginning

```
void insert_beginning(int item){
 Node *temp = new Node; //create new node
 temp->item=item;
 temp->next=head; //make next of new node to head
 head = temp;        //reset head to new node
}
```

Insertion at Middle



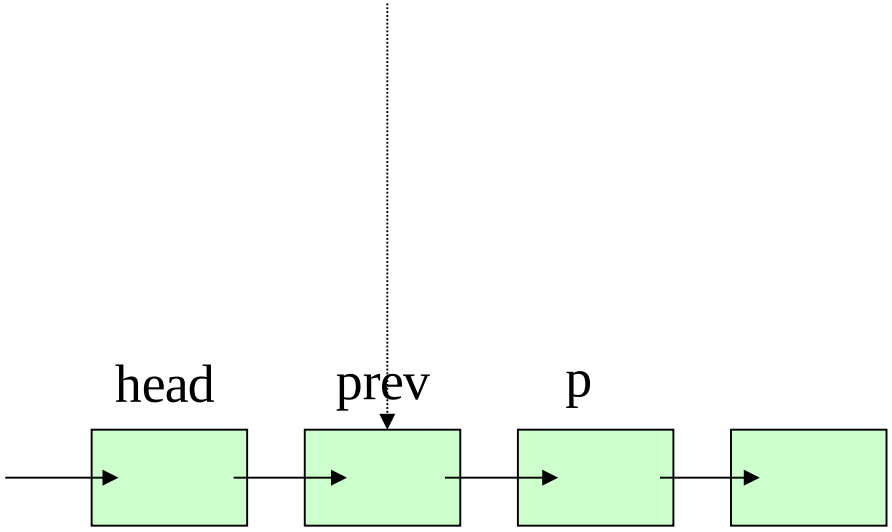
Insertion at Middle

```
void insert_after(int prev_item, int new_item) {
    Node *new_node = new Node; //create new node
    Node *temp; temp = head;
    while (temp != NULL) {
        if (temp->item == prev_item)
            break;
        else temp = temp->next;
    }
    if (temp != NULL) {
        new_node->item = new_item;
        new_node->next = temp->next;
        temp->next = new_node;
    } else {
                    //suitable message
    }
}
```

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 an item



Delete an item

```
void delete_item(int item) {
 Node *prev = NULL; Node *curr; curr=head;
 while(curr!=NULL) {
      if (curr->item == item) break;
      else { prev=curr; curr = curr->next;}
 }
 if(curr!=NULL){
   if(prev == NULL) head = head->next;
   else { prev->next=curr->next;}
   delete curr;
 }
    else {
    //suitable message
 }
}
```

Lists and Arrays

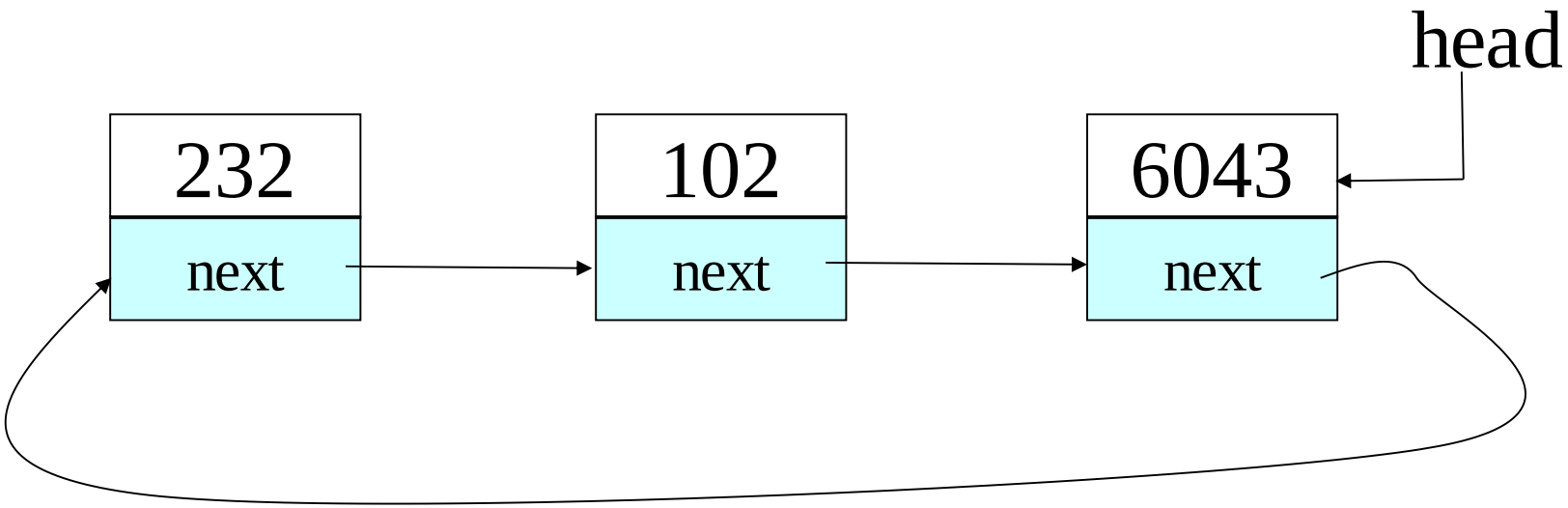
- Both represent a sequence.
- Continuity indicates next element in array; explicit indication in list.
- Therefore easy to change next element in a list.
- Insertion: only few references need to be changed; an array requires moving many elements.
- Random access of elements not possible unlike array.
- 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 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
  - is_empty()
  - 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.

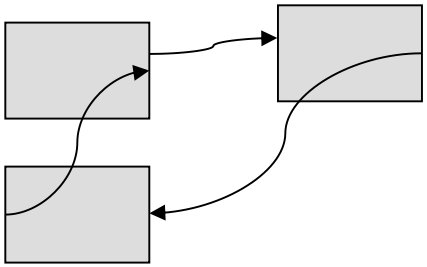
Circular Linked List

- Insertion at the Beginning

```
Node *new_node = new Node;
new_node->item=item;
if(head == null){head=new_node; // CLL was empty}
else new_node->next=head->next;
head->next=new_node;
```

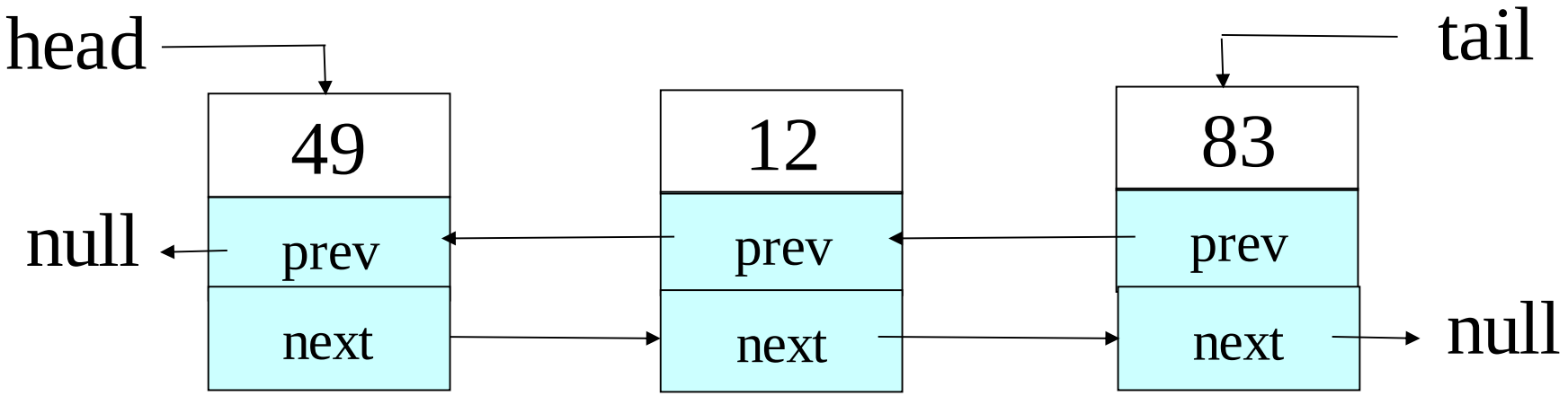
Circular Linked List

- 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
  - is_empty()
  - Insertion/Deletion beginning/end/middle of the list
  - Traversal
- Implementation
  - `structure 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.**

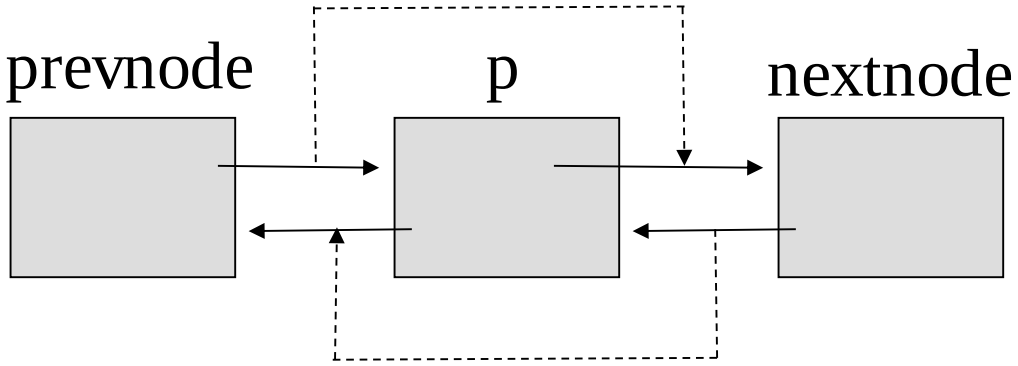
// p's previous node refer to p's next node

```
p->prev->next=p->next;
```

// p's next node refer to p's previous node

```
p->next->prev=p->prev;
```

// Special care for DLL with single node



Stack using LL

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

```
class Stack{
private:
  LinkedList* list;
public:
 Stack(){ list = new LinkedList; }
 bool is_empty(){ return list->is_empty(); }
 void push(int item){list->insert_beginning(item);
}
 int pop(){return list->delete_beginning(); }
}
```

Queue using LL

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

```
class Queue{
  private:
      LinkedList* list;
  Public:
      Queue() {list = new LinkedList;}
      bool is_empty(){return list->is_empty();}
      void insert(int item)    {list-
>insert_end(item);}
      int remove(){return list-
>delete_beginning();}
};
```

Applications

- Adding two big integers
  - 12313123131213123213 12 + 131231231231231
- Library management
- Sparse Table

Sparse Table: Problem Statement… Facts

- A university has 10, 000 students & 500 courses
- One student can take <= 5 courses per year
- One course can have <= 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

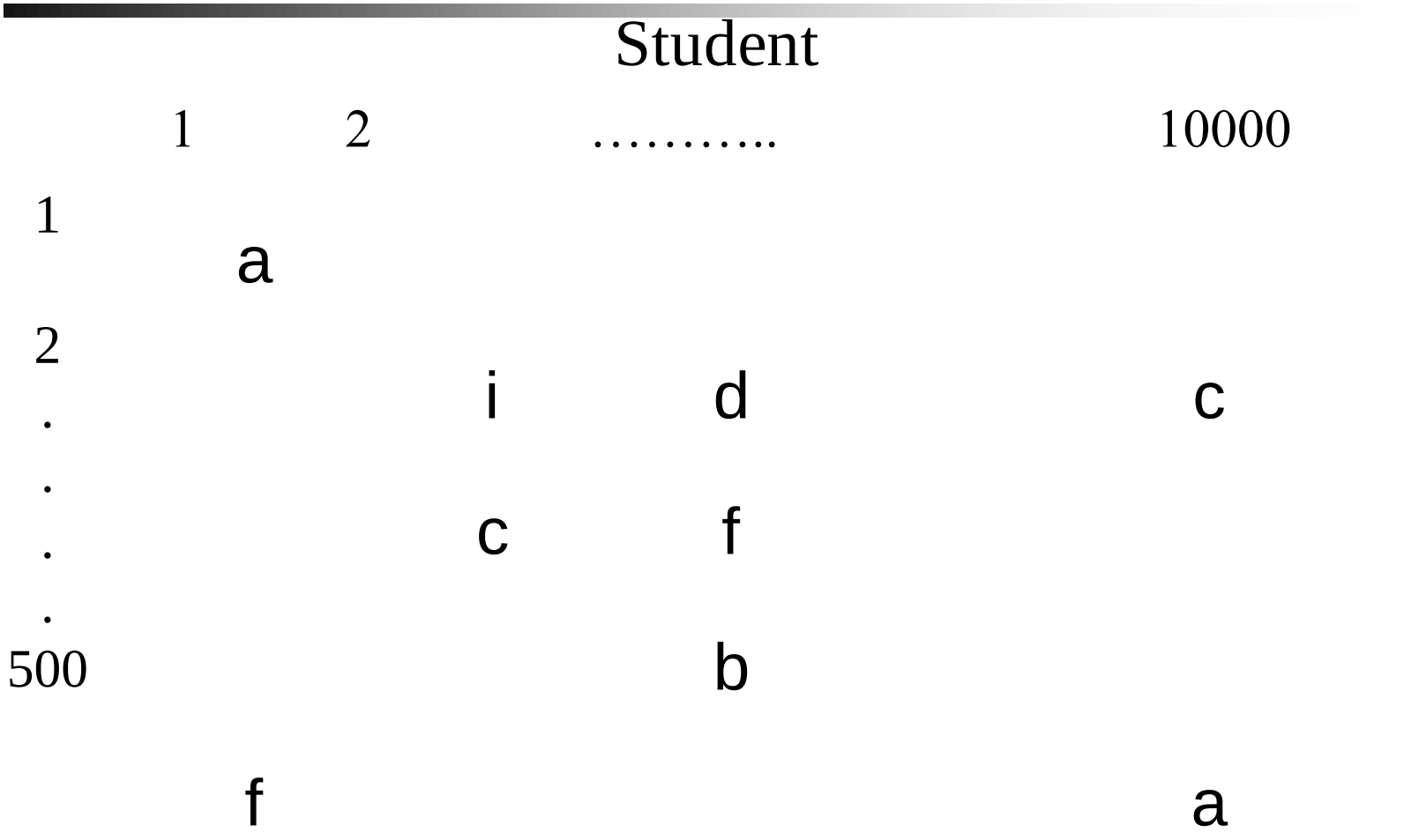
Sparse Table: Problem Statement…Constraints

- Time complexity
- space complexity

Sparse Table: Approaches

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

Sparse Table .. Approach 1



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
120, a			
	330, i	220, d	4900, c
	2540, c	3330, f	
		1750, b	
1400, f			4350, a

CRS-STD Table

Sparse Table .. Approach 2

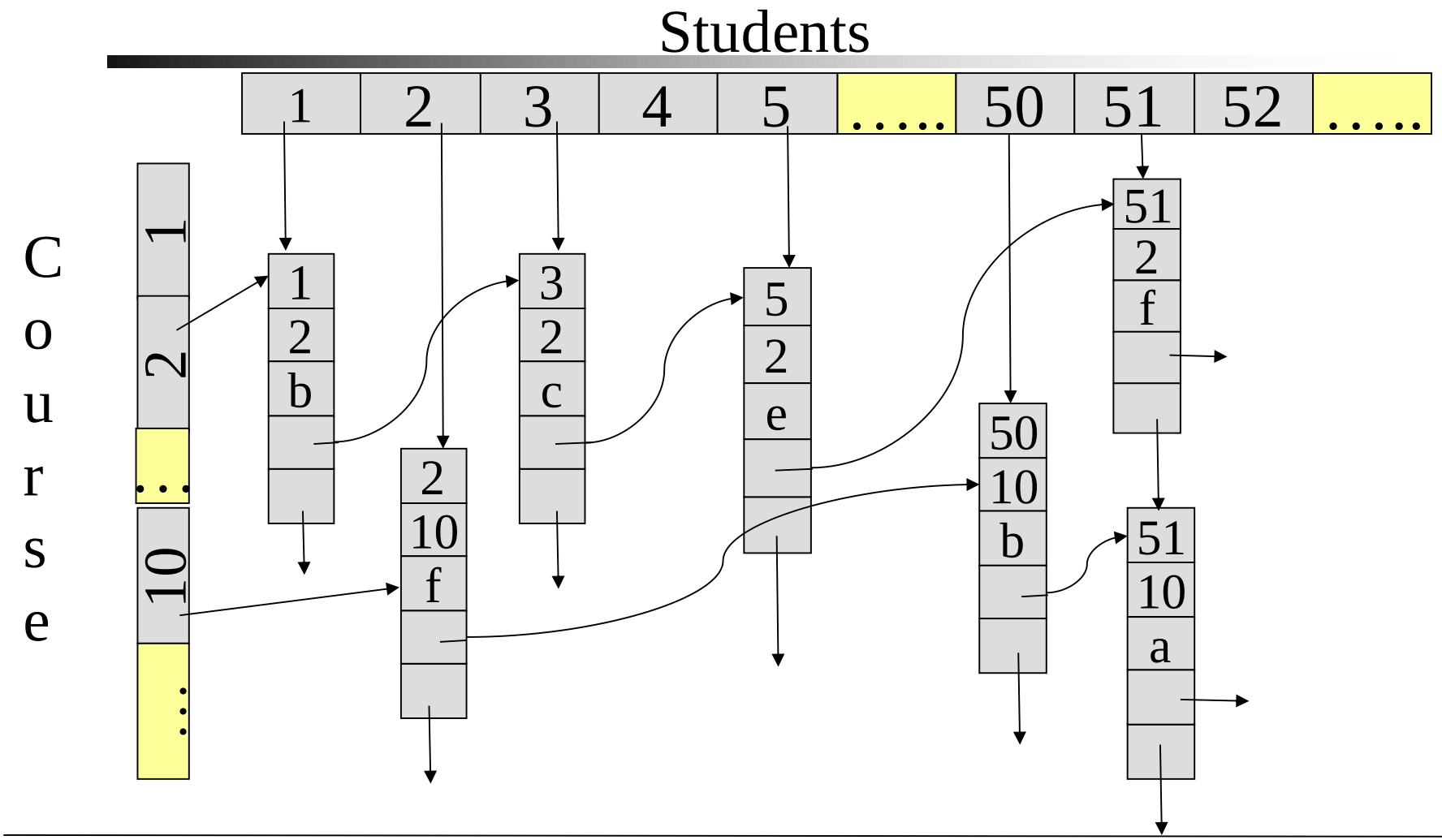
- Space required (assume 3 bytes/ cell)
  - Space(STD-CRS) + Space(CRS-STD) = 450KB
- Time complexity for the operations
  - List of students taking the course … from CRS-STD
  - List of courses taken by the student … from STD-CRS
- If one course allows say 500 students, every column must be of size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

Sparse Table .. Approach 3

- Linked List of courses
- Linked List of students

```
typedef struct StdCrsNode{
  int classId, stdId;
  int grade;
  StdCrsNode* nextCrs, nextStd;
}
```

Sparse Table .. Approach 3



Sparse Table .. Approach 3

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

Summary

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