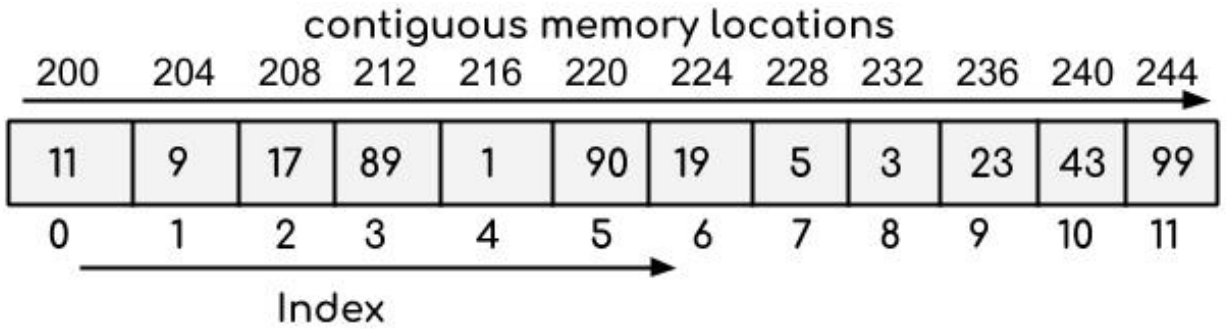
COMP2421 – DATA STRUCTURES AND ALGORITHMS

Linked Lists

Dr. Radi Jarrar
Department of Computer Science
Birzeit University

Data structure and Arrays

- A data structure is a way of storing data in a computer so that they can be retrieved and used efficiently
- An array is a very simple data structure for holding a sequence of data



Data structure and Arrays (2)

- Pros of Arrays
  - Access to an array element is fast since we can compute its location quickly
- Cons
  - Fixed size
  - When we want to insert or delete an element, we have to shift subsequent elements (slow)
  - We need a large enough block of memory to hold an array

Linked Lists

- Another data structure that is used to store sequence of data
- A linked list consists of a series of structures called nodes
- Data values do not have to be stored in adjacent memory cells
- Each node contains two fields: a "data" field and a "next" field, which is a pointer used to link one node to the next node
- To use a linked list, we only need to know where the first data value is stored

Linked Lists (2)

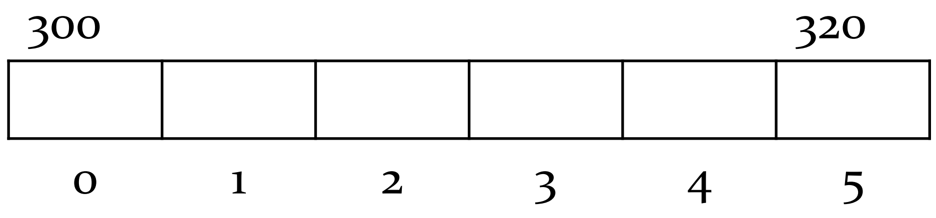
- Advantages of Linked Lists
  - Dynamic size
  - No shift of elements on deletion/insertion
- Drawbacks of Linked Lists
  - Random access isn’t allowed
  - Extra memory is needed for the next pointer

Linked Lists (3)

- When to use Linked Lists
  - The number of data items to be stored in the list is unknown
  - No need for random access
  - Insertion in the middle of the list is frequent

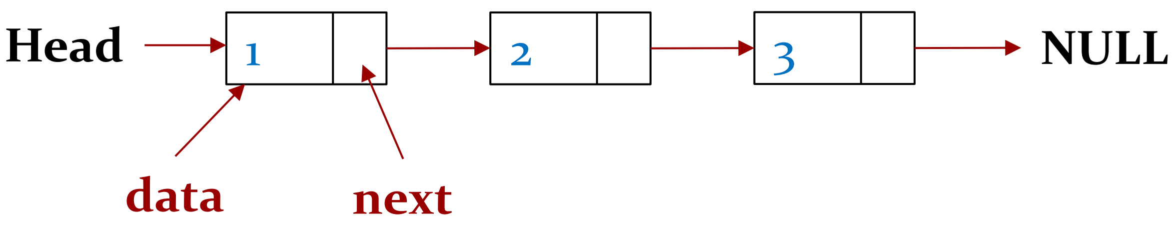
Array vs. Linked List

- *Cost of Accessing an element*
- Array



- Base address = 300
- Address of A[i] = 300 + i * 4
- Constant time O(1)

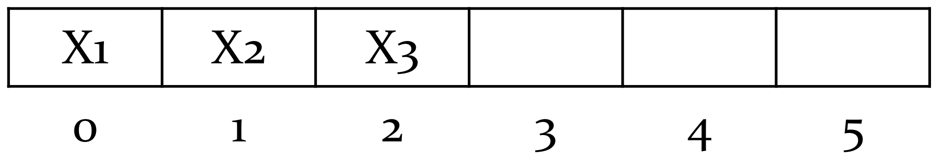
- Linked List



- Average case: O(n)

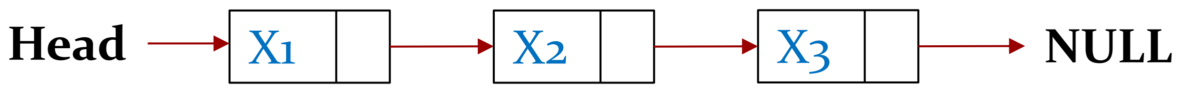
Array vs. Linked List

- *Memory requirements*
- Array



- Memory may not be available as one large block.

- Linked List

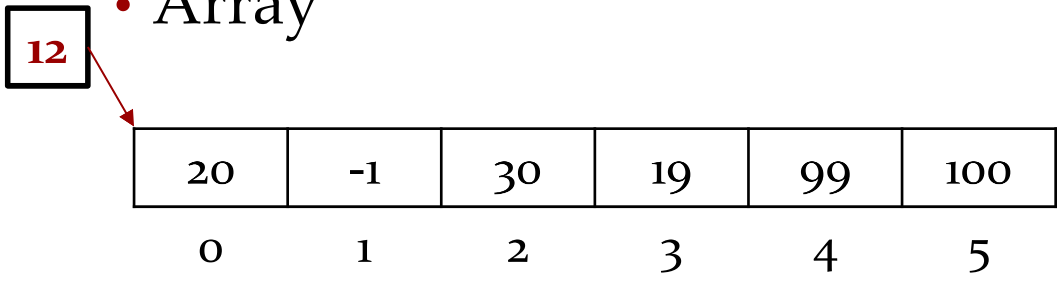


- No unused memory.
- Requires extra memory for pointer variables.
- Works well when memory may be available as multiple small blocks.

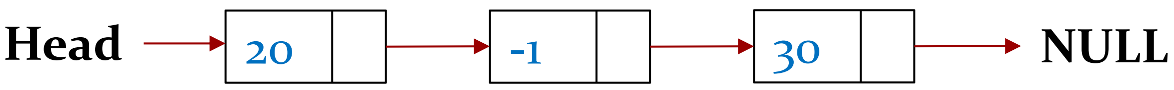
Array vs. Linked List

- *Cost of inserting/deleting an element*

- Array



- Linked List



- The cost of inserting/deleting a new element:
- At beginning → O(n)
- At end → O(1)
- At i^{th} position → O(n)

- O(1)
- O(n)
- O(n)

Linked Lists vs. Array

Operation	Array	Linked List
Print list		
Print Element		
Search		
Insert		
Delete		
Find Index		

Operations on Linked Lists

- Header node: a node that is kept at position zero. It points to the first element in the list.
- Creation (MakeEmpty): the process of creating the head node. Returns a pointer to the first node.
- Insertion: obtaining a new cell from the system by using the malloc call.
- Deletion: delete a given node after find.
- Find: search for a node. If exists, return a pointer to it.

Struct Node

- Node is the main building block of the list.
- In this example, each node contains a single data element and a pointer to the next node in the list.

```
struct node
{
    int Data;
    struct node* Next;
};
```

MakeEmpty

- Creates a Linked List

```
struct node* MakeEmpty(struct node* L){
        if(L != NULL)
                DeleteList( L );
        L = (struct node*)malloc(sizeof(struct node));

        if(L == NULL)
                printf("Out of memory!\n");

        L->Next = NULL;
        return L;
}
```

IsEmpty

- Checks if the list is empty

```
int IsEmpty(struct node* L){
    return L->Next == NULL;
}
```

IsLast

- Checks if a given node is the last node in the linked list

```
int IsLast(struct node* P, struct node* L){


}
```

Find

- Looks for a node in the Linked List. Returns a pointer to the node if exists.

```
struct node* Find(int X, struct node* L){
	struct node* P;
	P = L->Next;

	while(P != NULL && P->Data != X)
		X = X->Next;

	return P;
}
```

FindPrevious

- Similar to previous but return a pointer to the node previous to the one you are looking for. If X is not found, then Next field of returned value is NULL.

```
struct node* FindPrevious(int X, struct node* L){
    struct node* P;
    P = L;

    while(P->Next != NULL && P->Next->Data != X)
        P = P->Next;

    return P;
}
```

Delete

- Delete the first occurrence in the list. We find P, which is the cell pointer to the one containing X, via FindPrevious

```
void Delete(int X, struct node* L){
      struct node* P, temp;

      P = FindPrevious(X, L);

      if( !IsLast(P, L) ){
            temp = P->Next;
            P->Next = temp->Next; //bypass delete cell
            free(temp);
      }
}
```

Insert

- Pass an element to be inserted, a list L, and position P. Insert an element after the position implied by P.

```
void Insert(int X, struct node* L, struct node* P){
      struct node* temp;
      temp = (struct node*)malloc(sizeof(struct
node));
      temp->Data = X;
      temp->Next = P->Next;
      P->Next = temp;
}
```

PrintList

- Given a list, print its elements.

```
void PrintList(struct node* L){
	struct node* P = L;
	if( IsEmpty(L))
		printf("Empty list\n");
	else
		do{
			P=P->Next;
			printf("%d\t", P->Data);
		}while( !IsLast(P, L) );
		printf("\n");
}
```

DeleteList

- Given a list, delete all its elements.

```
void DeleteList(struct node* L){
	struct node* P, temp;
	P = L->Next;
	L->Next = NULL;

	while(P != NULL){
		temp = P->Next;
		free(P);
		P=temp;
	}
}
```

Size of Linked List

- Write a routine to find the size of a linked list.

```
int size( struct node* L){
    struct node* p = L->Next;
    int count = 0;
    while(p != NULL ){
        count += 1;
        p = p->Next;
    }
    return count;
}
```

Types of Linked Lists

- Linear singly-linked list
- Doubly linked list
- Single circular linked list
- Doubly circular linked list

Circular Linked List

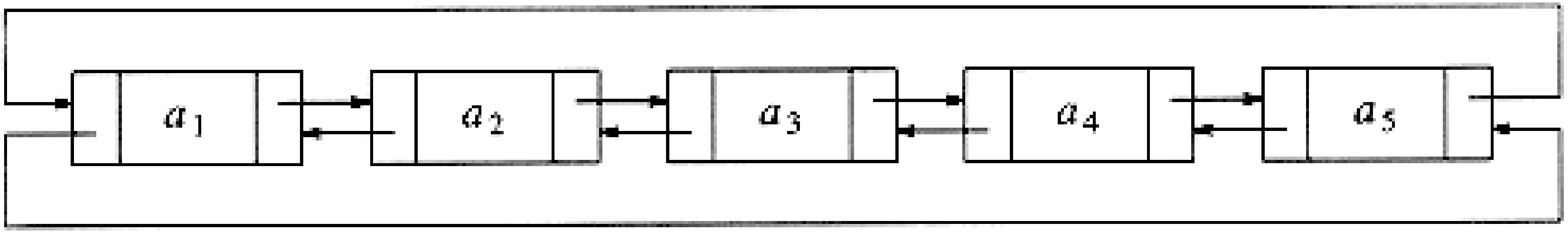
- The last node keeps a pointer to the first node

Doubly Linked List

- Each node points to its next and previous node
- Add an extra pointer to the previous node
- Adds more space requirements and doubles the cost of insertion & deletion because more pointers to fix
- Simplifies deletion-no need for FindPrevious

Doubly Circular Linked List

- Each node points to its next and previous node
- The last node's next is the first; and the previous of the first is the last



APPLICATIONS TO LINKED LISTS

Radix Sort

- Is a non-comparative sorting algorithm. We are not comparing elements (in a list for instance) with each other.

1. Takes the least significant digits (LSD) of the values to be sorted.
2. Sorts the list of elements based on the digit

https://youtu.be/7pwwgxmMHnc

Radix Sort (2)

- E.g., `9, 169, 739, 538, 10, 5, 36` → array size `7`
- Solution: consider 0 to 9 linked lists. 10 lists. Each one represent a digit which each significant digit can be. We are going to sort each number into one of these lists as we are going along.
  - Total of 10 lists
  - 0-9 refers to actual numbers

Radix Sort (3)

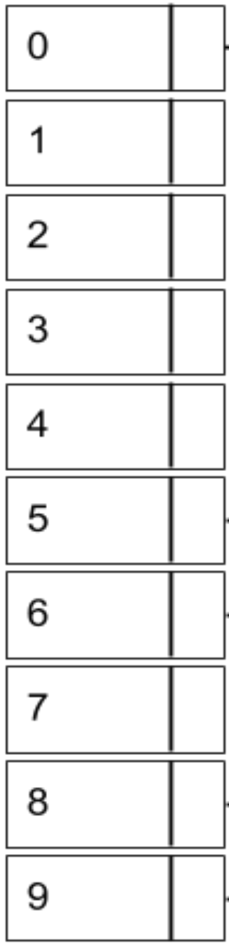
9	169	739	538	10	5	36

- STEP 1: take the least significant digit (the one's column). Extract using the mod `10` (`int m=10, n=1;`) (`m` is the modulus; divide the whole number, then divide the number by `n`).

Radix Sort (4)

9	169	739	538	10	5	36

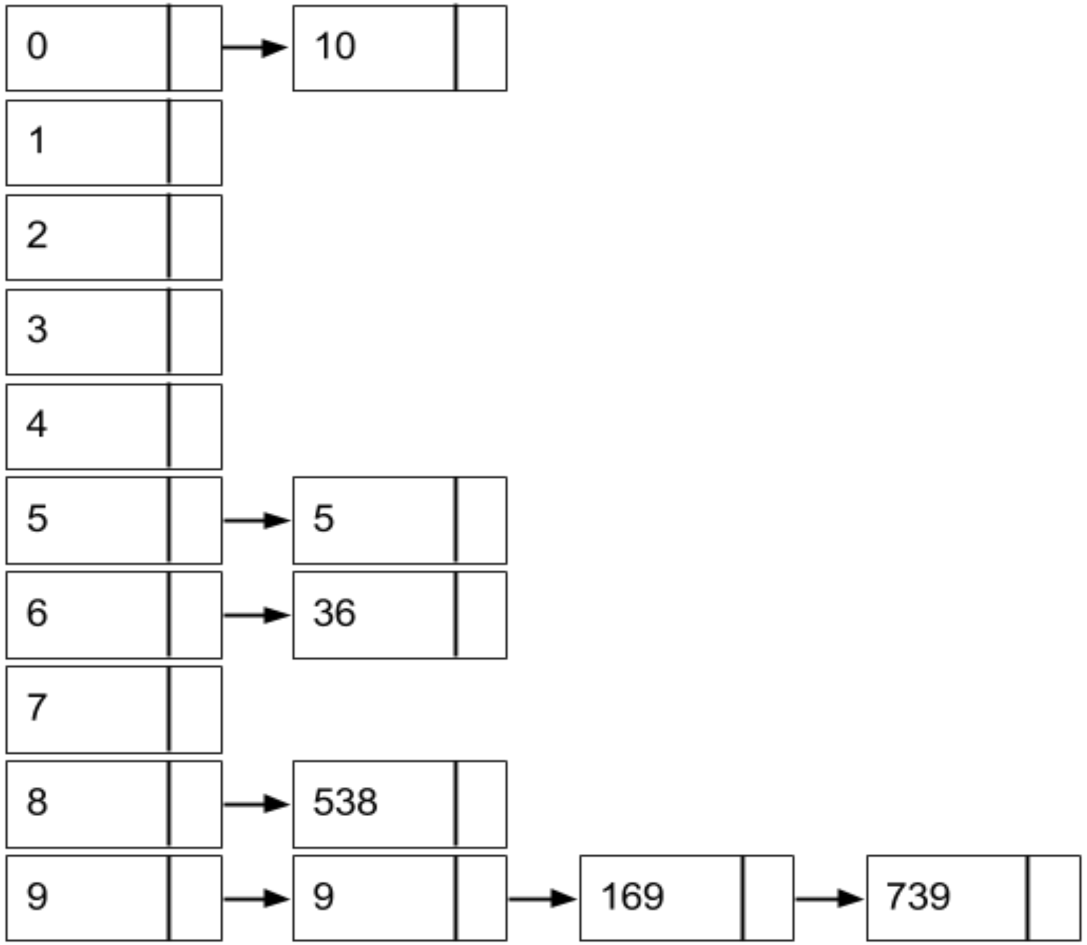
- So after the first round:



Radix Sort (4)

9	169	739	538	10	5	36

- So after the first round:



Radix Sort (5)

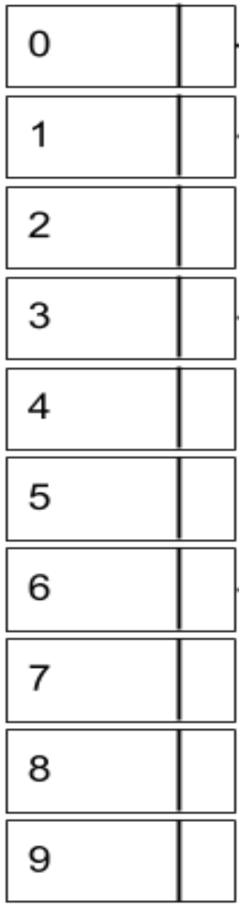
10	5	36	538	9	169	739

- Once we reached the end of the list, we make a new array and put the values by removing from head of each list.
- Then the sorted new array is: `10, 5, 36, 538, 9, 169, 739`
- Now we look at the second significant digit in the new array and we re-arrange the numbers based on that digit.
- Implementation (`m=m*10` (which is the mod); `n=n*10` which is `10` now)

Radix Sort (6)

10	5	36	538	9	169	739

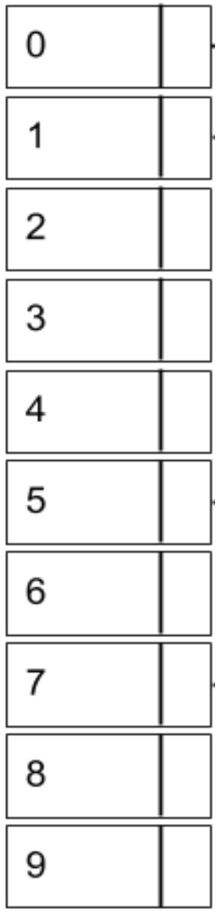
- Again, we take the mod of each number with m then we divide by `n` and put it in the list.



Radix Sort (7)

5	9	19	36	538	739	169

- So the list becomes `5, 9, 10, 36, 538, 739, 169`
- Now we look at the third digit:



Radix Sort (8)

- So the FINAL list becomes `5, 9, 10, 36, 169, 538, 739`
- Notes
  - The mod value m and the divisor value n go as big as the largest number of digits inside the array.
  - In other words, it increases one digit every time until array is sorted.
  - In this example, significant digit increase each time.

Radix Sort (9)

- Time complexity
  - O(kN) where N is the number of elements to sort, k is the number of digits (or it can be said for n keys which have d or fewer digits). Generally, k cannot be considered as a constant so it is not removed.
  - Best case: kN; average case: kN; worst case: kN

Radix Sort (10)

- Radix sort for strings?
- List of words: `dab, add, fee, bee, ace, eba`

Extra exercises on linked lists

- Question 1) Write a function that takes two sorted linked lists and return true if the lists are disjoin lists (meaning they have no common elements). Use iterations to solve this question.
- Question 2) Write a recursive function that takes two sorted linked lists and return true if the lists are disjoin lists (meaning they have no common elements). Your algorithm should be O(n).
- Question 3) Write a function to reverse a given doubly linked list.

Extra exercises on linked lists

- Question 4) Write a function called concat() that receives two lists and append the first one to the second.
- Question 5) Given a singly linked list, write a function to swap elements pairwise.

For example, if the linked list is 1->2->3->4->5 then the function should change it to 2->1->4->3->5, and if the linked list is 1->2->3->4->5->6 then the function should change it to 2->1->4->3->6->5.

Extra exercises on linked lists

- Question 6) Write a function called RemoveDuplicates() that takes a list sorted in increasing order and deletes any duplicate nodes from the list.

- Question 7) Write an iterative Reverse() function that reverses a list by rearranging all the .next pointers and the head pointer.