# ESc101: (Linear, Circular, Doubly) Linked Lists, Stacks, Queues, Trees

Instructor: Krithika Venkataramani

Semester 2, 2011-2012





## Introduction to Linked Lists

- Each bead connected to the next through a link
- Can change the order of the beads by changing the link/connection
- Bead ~ Data
- Linked beads ~ Linked list of data
- Changing links is useful in sorting
- Need not use additional temporary spaces as in array sorting

## Uses and Operations on Linked Lists

- Linear linked list: last element is not connected to anything
- Circular linked list: last element is connected to the first
- Dynamic: Size of a linked list grows or shrinks during the execution of a program and is just right
- Advantage: It provides flexibility in inserting and deleting elements by just re-arranging the links
- Disadvantage: Accessing a particular element is not easy
- There are three major operations on linked lists

1 Insertion

2 Deletion

3 Searching



## Structure for an element of the linked list

- A linked list contains a list of data
- The Data can be anything: number, character, array, structure, etc.
- Each element of the list must also link with the next element
- Therefore, a structure containing data and link is created
- The link is a pointer to the same type of structure

```
struct Node
{
int data ;
struct Node *next ;
};
```

- This is called a self-referential pointer

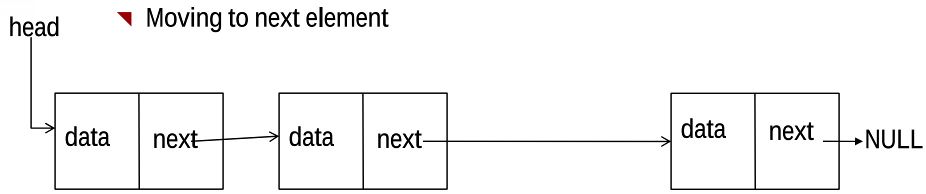
## Linked list: chain of nodes

- A linked list is simply a linear chain of such nodes
- The beginning of the list is maintained as a pointer to the first element (generally called head)
- Space for an element is created using a pointer (say q)
  - q = (struct Node *) malloc (size of (struct Node) );
  - q->data is the desired value
  - q->next is NULL
- A list element's members are accessed using the pointer (q) to the list element
  - data using q->data
  - next element pointer using q->next
- Moving to next element is done using pointers
  - q = q→next;



## Recap of Linear Linked Lists

- Each element: data + link (pointer to next element)
- Element is also called a "node"
- Head: address of first element
- Last element pointer: NULL
- All operations done using pointers
  - Allocation of space of element
  - Assigning and accessing data values
  - Moving to next element



## Linked List: element definition and creation

```
#include <stdio.h>
#include <stdlib.h>
typedef struct Node
{
        int data;       // data of a node: list is made of these elements
        struct Node *next;      // link to the next node
} node;
node *create_node(int val)
{
        node *n;
        n = malloc(sizeof(node));
        n->data = val;
        n->next = NULL;
        return n;
}
```



## Sample Linked List creation

| | | | | |
|---|---|---|---|---|
| | 0 | | ..... | |
| | .. | | 200 | 25 |
| p1 | 10 | 100 | 204 | NULL |
| head | 18 | 100 | | |
| p2 | 26 | 130 | | |
| end | 34 | 200 | | |
| p3 | 42 | 200 | | |
| | ... | | | |
| | 100 | 15 | | |
| | 104 | 130 | | |
| | --- | | | |
| | 130 | 20 | | |
| | | 200 | | |
| | --- | | | |

- node: int + pointer

```
node *p1, *head, *p2, *end, *p3;
p1 = create_node(15);
head = p1;
p2 = create_node(20);
/*insert at end*/
head->next = p2;
end = head->next;
p3 = create_node(25);
end->next = p3;
end = end->next;
```

## Insertion at the beginning of the list

- Create a new node (say q)
- Make q->next point to head
- Make head equal to q
- If list is empty, i.e., head is NULL
  - Make head equal to q



## Insertion at end of list

- Create a new node (say q)
- Find the last element (say p)
- Make p->next point to q
- If list is empty, i.e., head is NULL
  - Make head equal to q

## Deletion at the beginning of the list

- Make p equal to head
- Make head equal to head->next
- Delete p (by using free)
- If list is empty, i.e., head is NULL
  - Nothing to do
- If list contains only one element
  - Delete head
  - head is now NULL

Krithika Venkataramani (krithika@cse.iitk.ac.in)


## Deletion from the end of the list

- Find the last element (say p)
- While finding p, maintain q that points to p
  - q is the node just before p, i.e., q->next is p
- Make q->next NULL
- Delete p (by using free)
- If list is empty, i.e., head is NULL
  - Nothing to do
- If list contains only one element
  - Delete head
  - head is now NULL

Krithika Venkataramani (krithika@cse.iitk.ac.in)

## Searching a node (insert after, delete after)

- Make p equal to head
- While p->data not equal to the data that is being searched, make p equal to p->next
- Using search, insert after and delete after operations can be implemented
- Insert after p
  - Create a new node q
  - Make q->next equal to p->next
  - Make p->next equal to q
- Delete after p
  - Call the next node, i.e., p->next as q
  - Make p->next equal to q->next
  - Delete q

13

Krithika Venkataramani (krithika@cse.iitk.ac.in)

## Linked List: element definition and creation

```
#include <stdio.h>
#include <stdlib.h>
typedef struct Node
{
    int data;      // data of a node: list is made of these elements
    struct Node *next;     // link to the next node
} node;
node *create_node(int val)
{
    node *n;
    n = malloc(sizeof(node));
    n->data = val;
    n->next = NULL;
    return n;
}
```

14

Krithika Venkataramani (krithika@cse.iitk.ac.in)

## Displaying the data in the linked list

```
void print_list(node *h)
{   /*Display data in each element of the linked list*/
    node *p;
    p = h;
    while (p != NULL)
    {
        printf("%d --> ", p->data);
        p = p->next;
    }
}
```



## Inserting at end

```
int main()
{
    node *head = NULL;    // head maintains the entry to the list
    node *p = NULL, *q = NULL;
    int v = -1, a;
    printf("Inserting at end: Enter the data value:\n");
    scanf("%d", &v);
    while (v != -1)
    {
        q = create_node(v);
        if (head == NULL)
            head = q;
```

## Inserting at end (cont.)

```
else  /*non empty list*/
      {
            p = head;
            while (p->next != NULL)
                  p = p->next;
            p->next = q;
      }
      scanf("%d", &v);
 }
print_list(head);   /*Display the data in the list*/
```



## Inserting at the beginning

```
printf("Inserting at beginning\n");
scanf("%d", &v);
while (v != -1)
{
      q = create_node(v);
      q->next = head;
      head = q;
      scanf("%d", &v);
}
print_list(head); /*Display the data in the list*/
```

## Inserting after an element

```
printf("Inserting after\n");
 scanf("%d", &v);
 while (v != -1)
 {
        q = create_node(v);
        scanf("%d", &a);
        p = head;
        while ((p != NULL) && (p->data != a))
               p = p->next;
        if (p != NULL)
        {
               q->next = p->next;
               p->next = q;
        }
        scanf("%d", &v);
 }
 print_list(head); /*Display the data in the list*/
```



## Deleting from the end

```
printf("Deleting from end\n");
if (head != NULL)
 {
        p = head;
        while (p->next != NULL)
        {
               q = p;
               p = p->next;
        }
        q->next = NULL;
        free(p);
 }
 print_list(head); /*Display the data in the list*/
```

## Deleting from the beginning

```
printf("Deleting from beginning\n");

 if (head != NULL)
 {
        p = head;
        head = head->next;
        free(p);
 }
/*Empty list: i.e. head==NULL, do nothing*/
 print_list(head); /*Display the data in the list*/
```

Krithika Venkataramani (krithika@cse.iitk.ac.in)



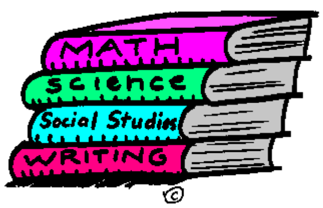
## Deleting after an element

```
  printf("Deleting after\n");
 scanf("%d", &a);
 p = head;
 while ((p != NULL) && (p->data != a))
        p = p->next;
 if (p != NULL)
 {
        q = p->next;
        if (q != NULL)
        {
               p->next = q->next;
               free(q);
        }
 }
 print_list(head); /*Display the data in the list*/
}
```

Krithika Venkataramani (krithika@cse.iitk.ac.in)

## Stacks and Queues

- The linked list only allows for sequential traversal
- Sequential traversal is present in stacks and queues
- Linked lists are used to implement these




Stack


Queue



Krithika Venkataramani (krithika@cse.iitk.ac.in)

## Stacks

- Insert at top of stack and remove from top of stack
- Stack operations also called Last-In First-Out (LIFO)
- Stack Operations: Push and Pop
- Push: insert at the top/beginning of stack
- Pop: delete from the top/beginning of stack



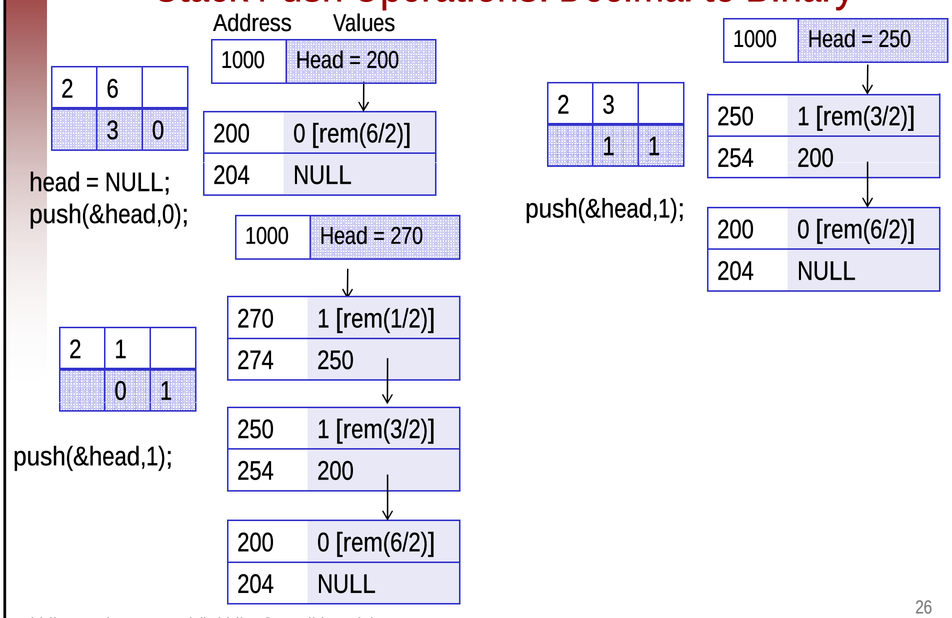
Krithika Venkataramani (krithika@cse.iitk.ac.in)

## Conversion of Decimal number to Binary

- Convert decimal number 39 to binary
- 39/2 = 19 +1
- 19/2 = 9 +1
- 9/2 = 4 +1
- 4/2 = 2 +0
- 2/2 = 1 +0
- 1/2 = 0 +1
- Read remainder from bottom to top
- Binary representation: 100111
- Stack can be used to read remainders in correct order



## Stack Push Operations: Decimal to Binary



## Stack Push

- stack top/head has the address of the first element
- Function needs the address to the stack top/head to make changes to head

```
void push(node **head_address, int top)
{
    node *q;
    q = create_node(top);  /*New element storing the new data*/
    q->next = *head_address; /*New element pointing to head*/
    *head_address = q; /*head pointing to new element*/
    return;
}
```


Krithika Venkataramani (krithika@cse.iitk.ac.in)

## Stack pop operations: Decimal to Binary

Head = 270

| | |
|---|---|
| 270 | 1 [rem(1/2)] |
| 274 | 250 |

| | |
|---|---|
| 250 | 1 [rem(3/2)] |
| 254 | 200 |

| | |
|---|---|
| 200 | 0 [rem(6/2)] |
| 204 | NULL |

rem= pop(&head); printf(“%d”, rem);

1

Head = 250

| | |
|---|---|
| 250 | 1 [rem(3/2)] |
| 254 | 200 |

| | |
|---|---|
| 200 | 0 [rem(6/2)] |
| 204 | NULL |

rem= pop(&head); printf(“%d”, rem);

11

Head = 200

| | |
|---|---|
| 200 | 0 [rem(6/2)] |
| 204 | NULL |

rem= pop(&head); printf(“%d”, rem);

110

Head = NULL


Krithika Venkataramani (krithika@cse.iitk.ac.in)

# Stack Pop

```
int pop(node **head_address)
{
    node *p, *head;
    int top;
    head = *head_address;   /*head has address of the first element*/
    if (head != NULL)
    {
        p = head;  //p: address of stack top element in stack
        top = p->data; //data in stack top/first element
        head = head->next; //head now has address of 2nd element in stack
        free(p); //remove the first element in stack
    }
    else
        top = -1; //-1 denotes invalid value or empty list
    *head_address = head;  /*reflect the changes to head outside*/
    return top;
}
```

29

Krithika Venkataramani (krithika@cse.iitk.ac.in)

# Stack operations: Decimal to Binary

```
void main()
{
    node *head = NULL;      // head: address of stack top or stack reference
    int decimal, rem, binary[20], j=0;
    printf("Enter the (positive) decimal value:");
    scanf("%d",&decimal);
    /*Push: store binary digits in correct order*/
    while(decimal>0)
    {
        rem = decimal%2;
        push(&head,rem);
        decimal = decimal/2;
    }
```

30

Krithika Venkataramani (krithika@cse.iitk.ac.in)

## Stack operations: Decimal to Binary (cont.)

```
    /*Pop : to read binary digits in correct order*/
    printf("Binary representation: ");
    while(head!=NULL)
    {
            rem = pop(&head);
            printf("%d",rem);
            binary[j]=rem;
            j++;
    }
    printf("\n");
    binary[j]= -1; //to denote end of binary representation
}
```



Krithika Venkataramani (krithika@cse.iitk.ac.in)

## Queues

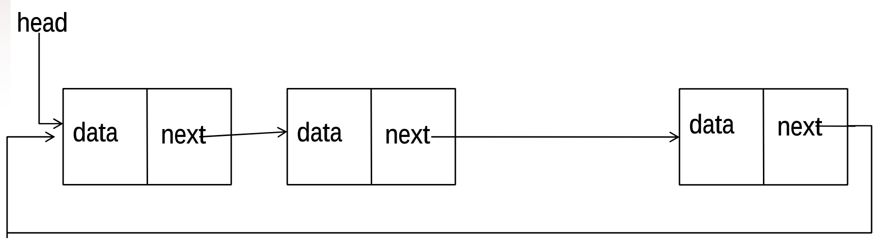
- Queue operations are also called First-in first-out
- Operations
  - Enqueue: insert at the end of queue
  - Dequeue: delete from the beginning of queue
- Code: similar to previous code on linked lists
- Queue Application: Executing processes by operating system
  - Operating System puts new processes at the end of a queue
  - System executes processes at the beginning of the queue



Krithika Venkataramani (krithika@cse.iitk.ac.in)

## Circular Lists

- The last element of a linked list points to the first element.
- A reference pointer is required to access the list: head

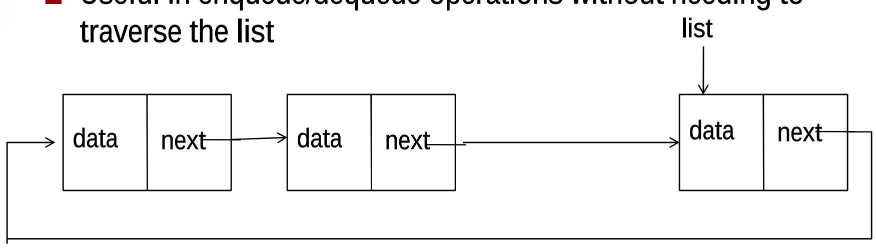




Krithika Venkataramani (krithika@cse.iitk.ac.in)

## Circular Lists

- The list pointer can have the address of the last element.
- The tail/last element can be accessed by the list pointer
- The head/first element can be accessed from the tail/last element (by list->next)
- Provides flexibility in accessing first and last elements
- Circular lists can be used for queues.
- Useful in enqueue/dequeue operations without needing to traverse the list





Krithika Venkataramani (krithika@cse.iitk.ac.in)

## Queue using a circular list

- Enqueue: insertion at the end of the list
  - The list pointer to the last element is known
  - Insert new element using this
- Dequeue: deletion at the beginning of the list
  - Traversing one element from the list pointer (to the last element) gives the first element
- Traversal of the entire list need not be done
- The list needs to be checked if it is empty

Krithika Venkataramani (krithika@cse.iitk.ac.in) 35

## Create an element in the queue

```
struct Node  //list element
{
    char *name;     // data of an element in the list
    struct Node *next;        // reference to the next element
};
struct Node *create_node(char *Name)  //create a list element
{
    struct Node *n;
    n = malloc(sizeof(struct Node)); //create space for  the element
    n->name = (char *)malloc((strlen(Name)+1)*sizeof(char)); /*create space for
    name*/
    strcpy(n->name,Name);
    n->next = NULL;
    return n;
}
```

Krithika Venkataramani (krithika@cse.iitk.ac.in) 36

## Print list of elements in queue

```
void print_list(struct Node *h)
{
        struct Node *p;
        p = h;
    if (p==NULL) //no element in list
        {
            printf("\nNo elements in the queue");
            return;
        }
    printf("Queue elements");
        p = p->next;    //first element in the list
    while (p!=h) //while last element has not been reached
        {
            printf("\n%s", p->name);
            p = p->next;
        }
        printf("\n%s", p->name); //print last element

}
```



## Enqueue: Add to the end of the queue

```
struct Node* enqueue(struct Node *list, char *Name)
{
  struct Node *n;
  n = create_node(Name); // create new element
  if (list==NULL)  // if no element in the queue
  {
        list = n;
        list->next = list;
  }
  else  //list points to the last element in queue
  {
        n->next = list->next; //give reference to first element from new element
        list->next = n; // add the new element to the end of queue
        list = n;  //provide reference to the end of the queue
  }
 return(list);
}
```

## Dequeue: Remove element from queue end

```
struct Node* dequeue(struct Node *list)
{
struct Node *temp;
 if (list==NULL) //error check
        printf("\nError: No elements in queue to dequeue");
 else if (list->next==list) //only one node
 {
        free(list); //return memory to system
        list = NULL;
 }
 else
 {
        temp = list->next; //first node
        list->next = list->next->next; //remove the link to the first node
        free(temp); //return memory of the deleted first node to the system
 }
 return(list);
}
```

Krithika Venkataramani (krithika@cse.iitk.ac.in) 39

## Calling different queue operations

```
void main()
{
        struct Node *list = NULL;    // address of the last element in the circular list
        char command, Name[50];
        scanf(" %c", &command);  //read queue operation
        while ((command!='S') && (command!='s')) //Stop operations: S
        {
                if ((command=='E') || (command=='e')) //Enqueue: E <name>
                {
                        scanf(" %s", Name);
                        list = enqueue(list, Name);
                }
                else if ((command=='D') || (command=='d')) //Dequeue: D
                        list = dequeue(list);
                else if ((command=='L') || (command=='l'))  //Print queue: L
                        print_list(list);
```

Krithika Venkataramani (krithika@cse.iitk.ac.in) 40

## Calling different queue operations (cont.)

```
        else  //error check
                printf("Incorrect operation");
        printf("\nEnter another queue operation: ");
        scanf(" %c", &command);
    }
}
```



## Josephus problem

- A set of players are present in a circle
- Counting from a given player, every ‘*n*th’ player is considered ‘out’ and eliminated from the game
- Counting starts again from the next person after the removed player, and the next ‘*n*th’ player is removed.
- The game continues until only one player remains, who is the winner

## Algorithm for Josephus Problem

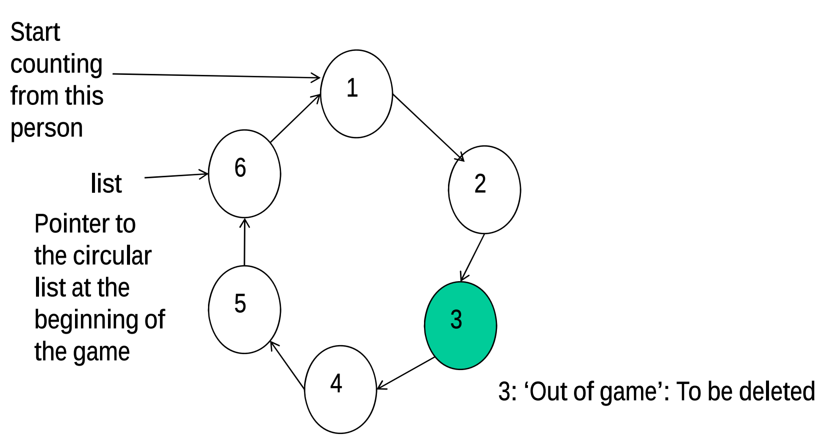
1. Obtain the initial player list
2. Go to starting player. Start count of 1.
3. Increment count, go to next player. If player-list end is reached, go to list beginning.
4. If count < n, go back to step 3
5. If count = n, remove player from list. Set count = 1 from next player. Go back to Step 3.
6. If next player is same as current player, declare winner.

- Implementation
  - 2D Arrays to hold player names
  - Circular lists
- Circular lists are an easier implementation for Step 3
  - Step 5: easier with doubly linked circular lists
  - workaround: eliminate nth player when count = n-1

43

Krithika Venkataramani (krithika@cse.iitk.ac.in)

## Josephus Problem with n = 3, starting from '1'

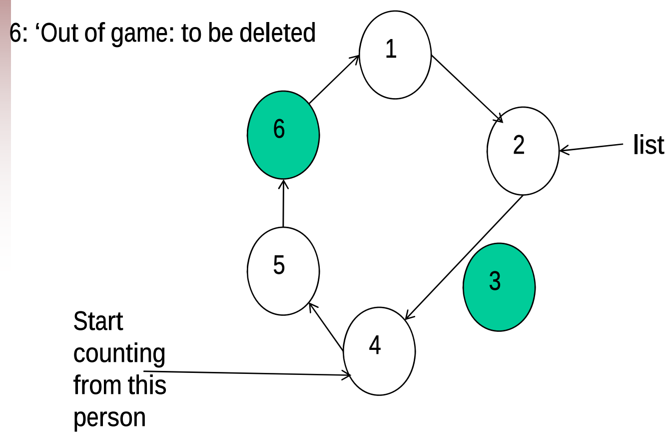



List pointer: to the end of list

44

Krithika Venkataramani (krithika@cse.iitk.ac.in)

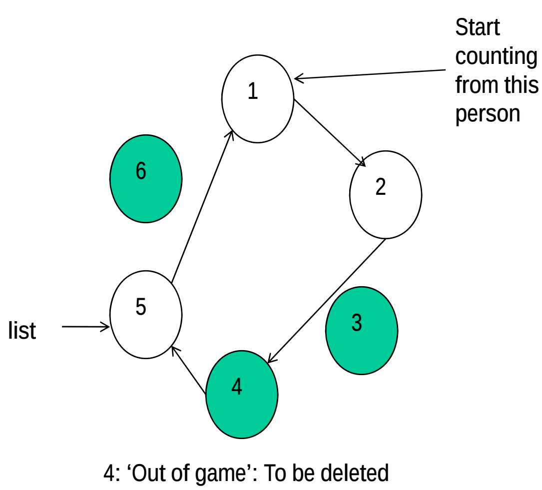
## Josephus Problem with n = 3, starting from '1' (cont.)




Krithika Venkataramani (krithika@cse.iitk.ac.in)

45

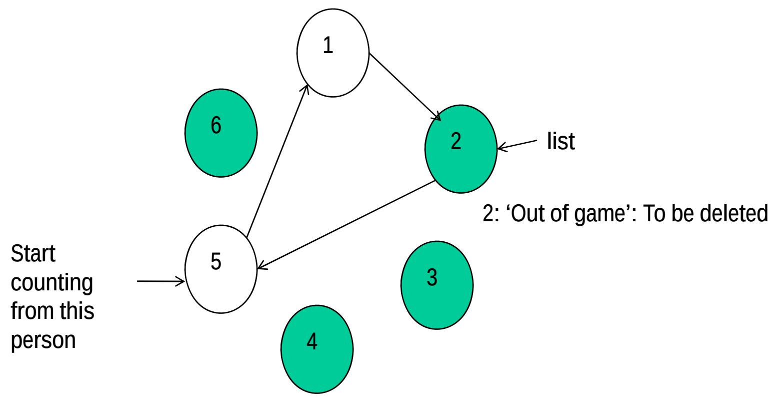
## Josephus Problem with n = 3, starting from '1' (cont.)




Krithika Venkataramani (krithika@cse.iitk.ac.in)

46

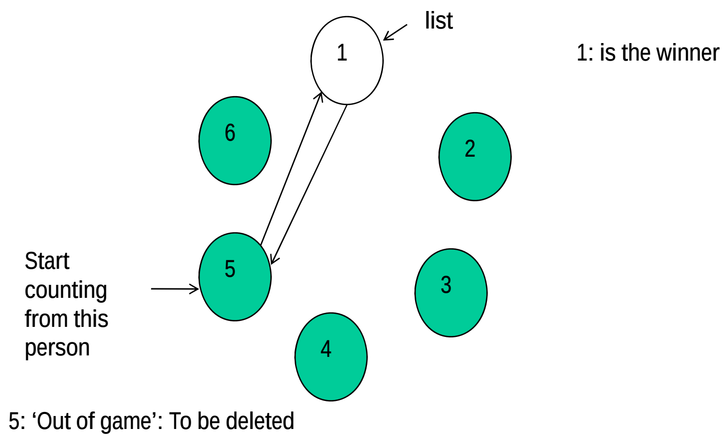
# Josephus Problem with n = 3, starting from '1' (cont.)




Krithika Venkataramani (krithika@cse.iitk.ac.in)

47

# Josephus Problem with n = 3, starting from '1' (cont.)




Krithika Venkataramani (krithika@cse.iitk.ac.in)

48

## Create an element in the list

```
struct Node  //list element
{
     char *name;     // data of an element in the list
     struct Node *next;        // reference to the next element
};
struct Node *create_node(char *Name)  //create a list element
{
     struct Node *n;
     n = malloc(sizeof(struct Node)); //create space for  the element
     n->name = (char *)malloc((strlen(Name)+1)*sizeof(char)); /*create space for
     name*/
     strcpy(n->name,Name);
     n->next = NULL;
     return n;
}
```



## Enqueue: Add to the end of the list

```
struct Node* enqueue(struct Node *list, char *Name)
{
  struct Node *n;
  n = create_node(Name); // create new element
  if (list==NULL)  // if no element in the queue
  {
          list = n;
          list->next = list;
  }
  else  //list points to the last element in queue
  {
          n->next = list->next; //give reference to first element from new element
          list->next = n; // add the new element to the end of queue
          list = n;  //provide reference to the end of the queue
  }
  return(list);
}
```

## Dequeue: Remove element from queue beginning

```
struct Node* dequeue(struct Node *list)
{
struct Node *temp;
 if (list==NULL) //error check
        printf("\nError: No elements in queue to dequeue");
 else if (list->next==list) //only one node
 {
        free(list); //return memory to system
        list = NULL;
 }
 else
 {
        temp = list->next; //first node
        list->next = list->next->next; //remove the link to the first node
        free(temp); //return memory of the deleted first node to the system
 }
 return(list);
}
```

Krithika Venkataramani (krithika@cse.iitk.ac.in) 51

## Josephus problem code

```
void josephus()
{
    char Name[50], *end = "end";
    struct Node  *list = NULL; // 'Node' data has player name. 'list' points to
end of the list
    int n, i;
    printf("\nEnter list of player names\n");
    scanf(" %s", Name);
    while (strcmp(Name,end)!=0)   /*create the list of players, reading names
until "end"*/
    {
        list = enqueue(list, Name); /*same  enqueue function as before to
create list*/
        scanf(" %s", Name);
    }
    printf("Enter the count of the next player eliminated: ");
    scanf("%d",&n);  //nth player eliminated
```

Krithika Venkataramani (krithika@cse.iitk.ac.in) 52

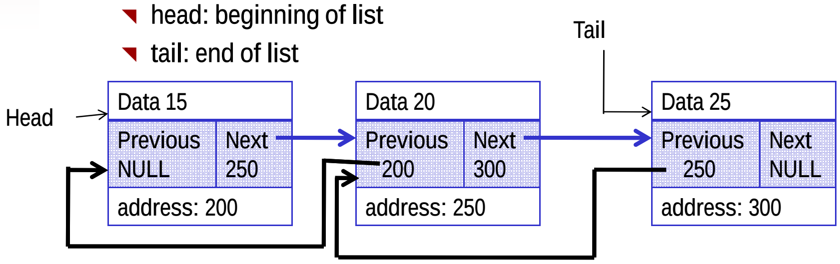
## Eliminating players

```
printf("Order of players eliminated from game");
/* Play the game by starting count from list beginning*/
 while (list !=list->next) // while more than one player is left, continue game
 {
       for (i=1; i < n; i++)
             list = list->next;
       printf("\n%s",list->next->name); // name of the player eliminated from the game
       list = dequeue(list); //same dequeue function as before
 }
 printf("The winner of the game is: %s", list->name);
}
```

Krithika Venkataramani (krithika@cse.iitk.ac.in) 53

## Doubly linked lists

- Linked list disadvantages
  - Cannot traverse the list backwards
  - Cannot delete an element using only a pointer to that element
- Doubly linked lists: pointers to next element as well as previous element
  - Use previous element pointer for given node deletion
- Pointers to both ends of the lists are stored
  - head: beginning of list
  - tail: end of list




Krithika Venkataramani (krithika@cse.iitk.ac.in) 54

## Doubly linked lists: applications

- Queue implementation with doubly linked lists
  - Enqueue: insertion at Queue end (using tail )
  - Dequeue: deletion at Queue begin (using head)
- Addition of long integers through doubly linked lists
  - Traverse from list end while adding
  - Traverse from list beginning to display


Krithika Venkataramani (krithika@cse.iitk.ac.in)

## Doubly Linked List: element definition

```
typedef struct Node {       /*Element definition*/
    int          data;
    struct node  *previous;
    struct node  *next;
} node;
node  *head = NULL;
node  *tail = NULL;
node *s; /*Creating first node*/
s = (node *) malloc (sizeof (node));
s->data = 15;
s->previous = NULL;
s->next = NULL;
head = s;
tail = s;
```


Krithika Venkataramani (krithika@cse.iitk.ac.in)

## Removing an element

```
printf("Deleting a given element\n");
  scanf("%d", &a); /*enter the data to be deleted*/
  p = head;
  while ((p != NULL) && (p->data != a)) /*searching the data to be deleted*/
    p = p->next;
  if (p != NULL) /*p is the node to be deleted*/
  {
    if (p->previous == NULL) /*if p is head*/
    {
              head = head->next; /*move head to next element*/
              head->previous = NULL:
    }
```



## Removing an element (cont.)

```
  else if (p->next == NULL) /*if p is tail*/
  {
              tail = tail->previous; /*move tail to previous element*/
              tail->next = NULL;
  }
  else /*p is in the middle of the list*/
    {
              (p->previous)->next = p->next;
              (p->next)->previous = p->previous;
    }
    free(p);   //deleting the node p
  }  /*end of if statement*/
print_list(head);
```

## Inserting after a given element

- Will not change head
- May change tail, if the given element (n) is the last element

```
q = create_node(v); //new node q
scanf("%d", &a); //enter data value, a, to be searched for
p = head;
while ((p != NULL) && (p->data != a)) //a is being searched for
        p = p->next;
if (p != NULL)  //new node q is inserted after p
{
        q->previous = p;
        q->next = p->next;
        if (p->next==NULL) //if p is tail
                tail = q; //q becomes new tail
        else
            (p->next)->previous = q;
        p->next = q;
}
```

Krithika Venkataramani (krithika@cse.iitk.ac.in) 59

## Binary Trees

- Searching is more efficient in a binary tree than from an array
  - Binary search can be easily understood using binary trees
- Searching starts from the top/root of the binary search trees
- Search stops if the number, n, is found in the node
- If n < the number stored in the node, searching is done from the left child
- If n > the number stored in the node, searching is done from the right child
- The procedure is continued till the number is found.

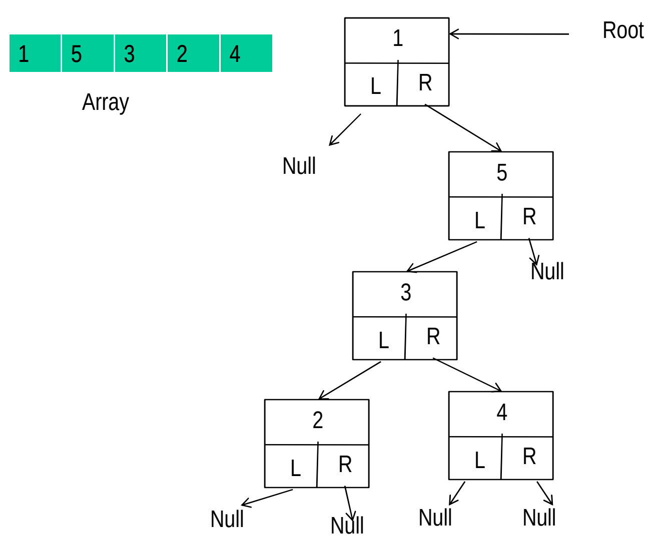
Krithika Venkataramani (krithika@cse.iitk.ac.in) 60

## Binary Tree Creation

- A binary tree can be created from an array
- One approach:
  - Use the first element as the root node, R.
  - If the 2nd element is less (greater) than the root, insert it as a left (right) child, C
  - If the 3rd element is less(greater) than the root, traverse to the left (right) child, C, of root node
  - If the 3rd element is less (greater) than this child node, insert it as a left (right) child to C.
  - Repeat this process for every element in the array
- If the array is sorted, this approach will lead to an 'unbalanced' tree



## Binary Tree Creation



The content of some of these slides are from the lecture slides of Prof. Arnab Bhattacharya and Prof. Dheeraj Sanghi

63

Krithika Venkataramani (krithika@cse.iitk.ac.in)