Review of Elementary Data Structures (Part 1)

Manoj Kumar
DTU, Delhi

What is data structure?

- In computer science, a **data structure** is a particular way of storing and organizing data in a computer so that it can be used efficiently.
- A data structure is a way of organizing data that considers not only the items stored, but also their relationship to each other. Advance knowledge about the relationship between data items allows designing of efficient algorithms for the manipulation of data.

Data structures

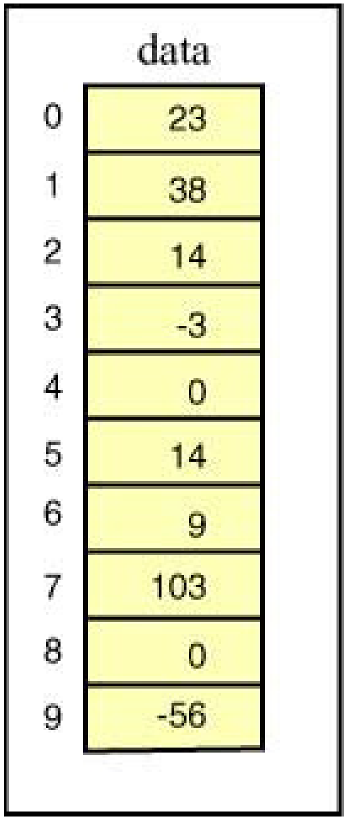
- Data structures provide a means to manage huge amounts of data efficiently.
- Usually, efficient data structures are a key to designing efficient algorithms.
- Some formal design methods and programming languages emphasize data structures, rather than algorithms, as the key organizing factor in software design.

ARRAYS

- A collection of data elements in which
  - all elements are of the same data type, hence homogeneous data
    - An array of students' marks
    - An array of students' names
    - An array of objects (OOP perspective!)
  - elements (or their references) are stored at contiguous/ consecutive memory locations
- Array is a static data structure
  - An array cannot grow or shrink during program execution – its size is fixed

ARRAY: Basic Concepts

- Array name (data)
- Index/subscript (0...9)
- The slots are numbered sequentially starting at zero (Java, C, C++)
- If there are N slots in an array, the index will be 0 through N-1
  - Array length = N = 10
  - Array size = N x Size of an element
    = 10x2 = 20 bytes
- Direct access to an element



ARRAY: Memory representation

- **Starting address of array:** 10266 (base address)
- **Size of each element:** 2 (for integer array)
- **Length of array :** n=6
- **Total memory occupied by array:** 6x2=12 bytes.
- **Address of A[0]:** 10266
- **Address of A[1]:** 10268
- **Address of A[i]:** base address + i * size of element = 10266 + i*2

Address	Value	
10266	45	A[0]
10268	52	A[1]
10270	23	A[2]
10272	54	A[3]
10274	12	A[4]
10276	6	A[5]

ARRAY: 2D array

	Col 0	Col 1	Col 2	…	Col n_2-1
Row 0	**A[0][0]**	**A[0][1]**	**A[0][2]**	**…**	**A[0][n_2-1]**
Row 1	**A[1][0]**	**A[1][1]**	**A[1][2]**	**…**	**A[1][n_2-1]**
…	**…**	**…**	**…**	**…**	**…**
Row n_1-1	**A[n_1-1][0]**	**A[n_1-1][1]**	**A[n_1-1][2]**	**…**	**A[n_1-1][n_2-1]**

$A[n_1][n_2]$

ARRAY: 2D array representation (in Memory)

25	28	67	89
11	34	65	78
62	21	43	51

Array A[3][4]

	Address		
Row- 0	10266	25	A[0][0]
	10268	28	A[0][1]
	10270	67	A[0][2]
	10272	89	A[0][3]
Row-1	10274	11	A[1][0]
	10276	34	A[1][1]
	10278	65	A[1][2]
	10280	78	A[1][3]
Row-2	10282	62	A[2][0]
	10284	21	A[2][1]
	10286	43	A[2][2]
	10288	51	A[2][3]

Row-major Form

	Address		
Col- 0	10266	25	A[0][0]
	10268	11	A[1][0]
	10270	62	A[2][0]
Col-1	10272	28	A[0][1]
	10274	34	A[1][1]
	10276	21	A[2][1]
Col-2	10278	67	A[0][2]
	10280	65	A[1][2]
	10282	43	A[2][2]
Col-3	10284	89	A[0][3]
	10286	78	A[1][3]
	10288	51	A[2][3]

Column-major Form

SAMSUNG

DTU Delhi Technological University

ARRAY: 2D array representation (in Memory)

	Col-0	Col-1	Col-2	Col-3
Row-0	**25**	**28**	**67**	**89**
Row-1	**11**	**34**	**65**	**78**
Row 2	**62**	**21**	**43**	**51**

- Starting address of Array = Address of A[0][0] := 10266 (Base Address)
- Array dimension: $n_1 * n_2$:= 3*4
- Size of one element = s=2 bytes (integer array)
- Address of A[i][j] =

  base address + (number of elements before A[i][j]) * size of element)
- i.e. Address of A[i][j] = base address + $(i * n_2 + j) * s$
- Example: Address of A[1][3] = 10266 + (1 * 4 + 3) * 2

  = 10266 + 14

  = 10280

	Address		
Row- 0	10266	**25**	A[0][0]
	10268	**28**	A[0][1]
	10270	**67**	A[0][2]
	10272	**89**	A[0][3]
Row-1	10274	**11**	A[1][0]
	10276	**34**	A[1][1]
	10278	**65**	A[1][2]
	10280	**78**	A[1][3]
Row-2	10282	**62**	A[2][0]
	10284	**21**	A[2][1]
	10286	**43**	A[2][2]
	10288	**51**	A[2][3]

Row-major Form

ARRAY: Column-major form

- Address of A[i][j] =
  - base address + number of elements before A[i][j]) * size of element
- i.e. Address of A[i][j] = base address + (j * n_1 + i) * s

ARRAY: array representation

- Let Array is m-dimensional :$A[n_1][n_2]...[n_m]$
- Starting address of Array = Base Address
- Array dimension: $n_1 * n_2 * ... * n_m$
- Size of one element = s
- Address of $A[i_1][i_2]...[i_m]$ =

  base address + (number of elements before

  $A[i_1][i_2]...[i_m]$) * size of element)

$= \text{base address} + (i_1 * n_2 * n_3 * ... * n_m + i_2 * n_3 * n_4 * ... * n_m +$

$... + i_{m-1} * n_m + i_m) * s$

ARRAY: Operations

- **Indexing**: inspect or update an element using its index. Performance is very fast O(1)
  - randomNumber = numbers[5];
  - numbers[20000] = 100;
- **Insertion:** add an element at certain index
  - Start: very slow O(n) because of shift
  - End : very fast O(1) because no need to shift
- **Removal:** remove an element at certain index
  - Start: very slow O(n) because of shift
  - End : very fast O(1) because no need to shift
- **Search:** performance depends on algorithm
  - 1) Linear: slow O(n) 2) binary : O(log n)
- **Sort:** performance depends on algorithm
  - 1) Bubble: slow $O(n^2)$ 2) Selection: slow $O(n^2)$
  - 3) Insertion: slow $O(n^2)$ 4) Merge : O (n log n)

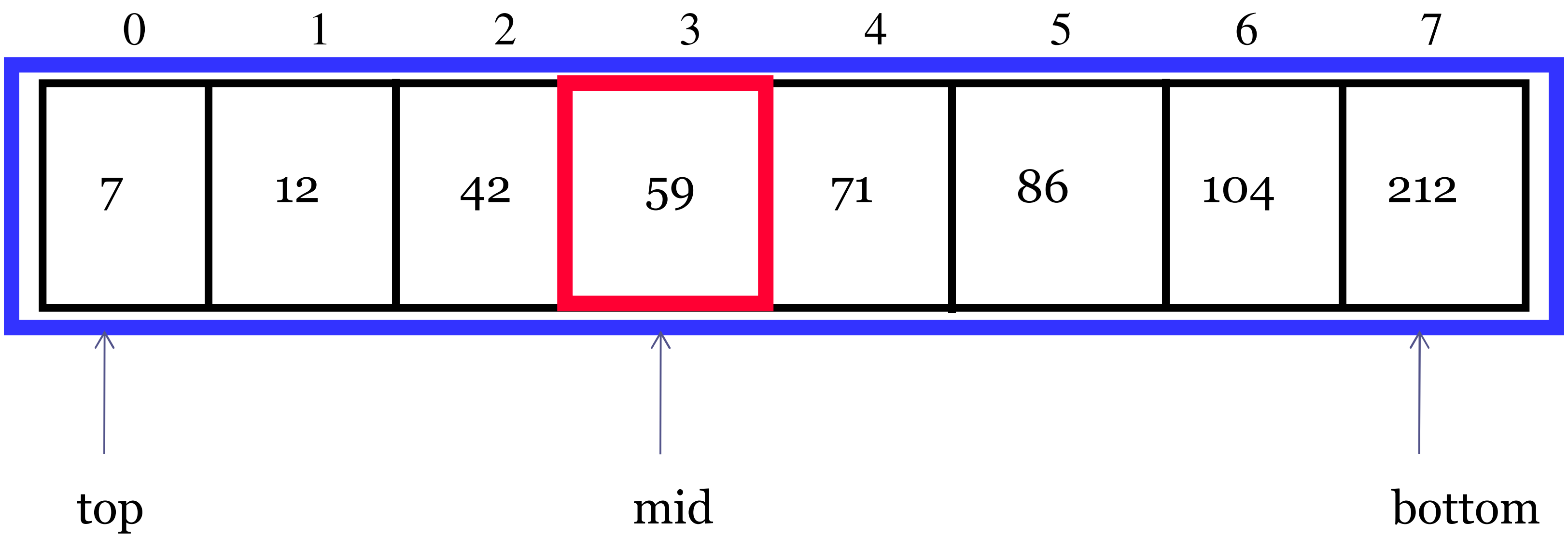
Unsorted Array: Operations

- Find (search) an element
  - O(N)   Linear search
- Find the smallest/largest element
  - O(N)
- Insert an element (at end)
  - O(1)
- Insert an element (at start)
  - O(N)
- Remove an element (from end)
  - O(1)
- Remove an element (from start)
  - O(N)

Sorted Array: Operations

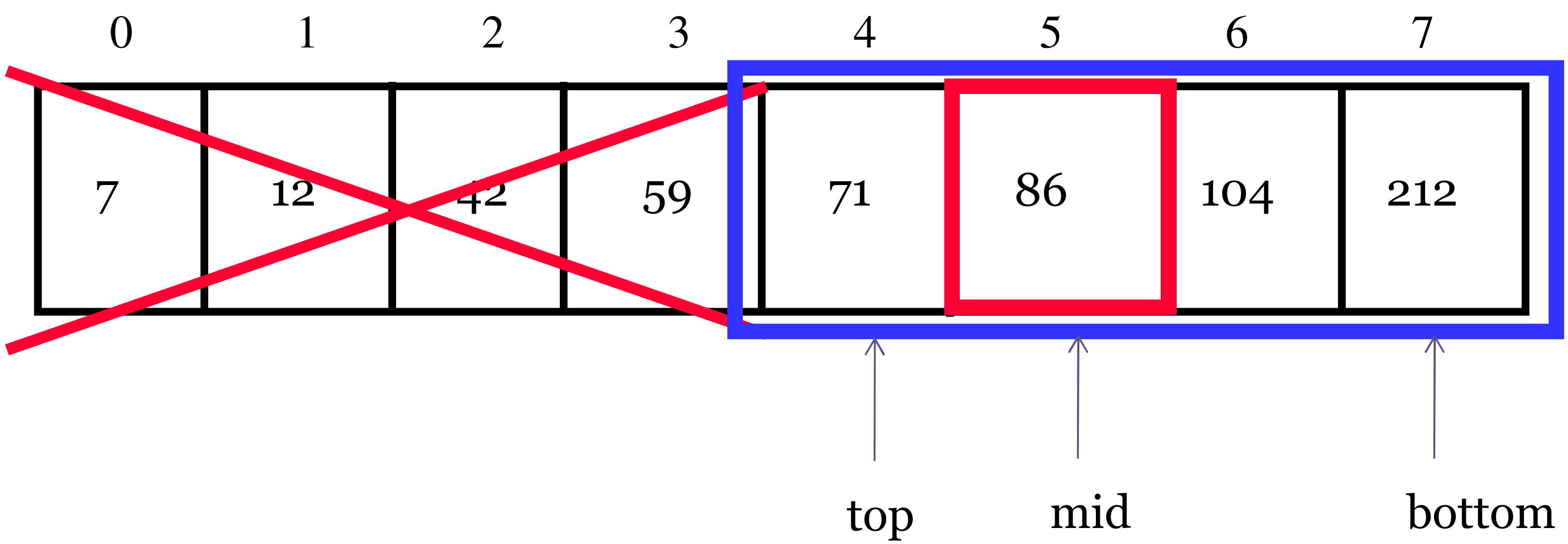
- Find (search) an element
  - O(lg N)   Binary search
- Find the smallest/largest element
  - O(1)
- Insert an element (at end)
  - O(1)
- Insert an element (at start)
  - O(N)
- Remove an element (from end)
  - O(1)
- Remove an element (from start)
  - O(N)

Binary Search Example



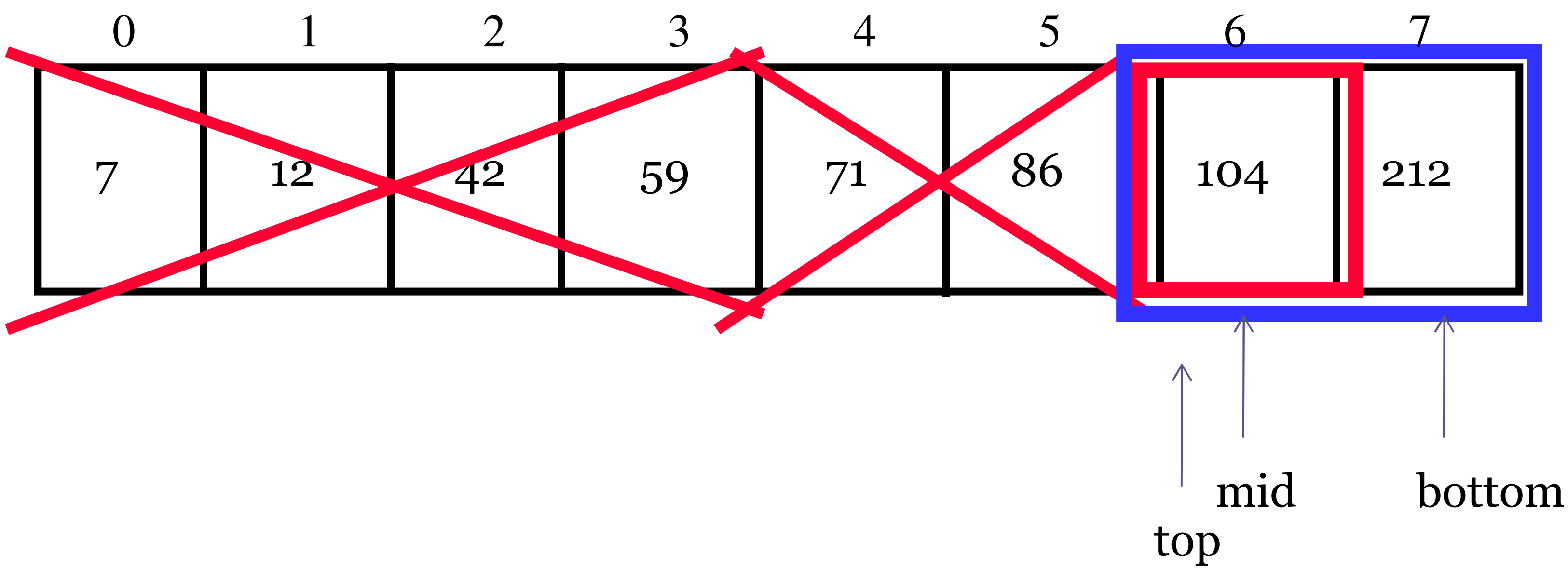
Looking for 89

Binary Search Example



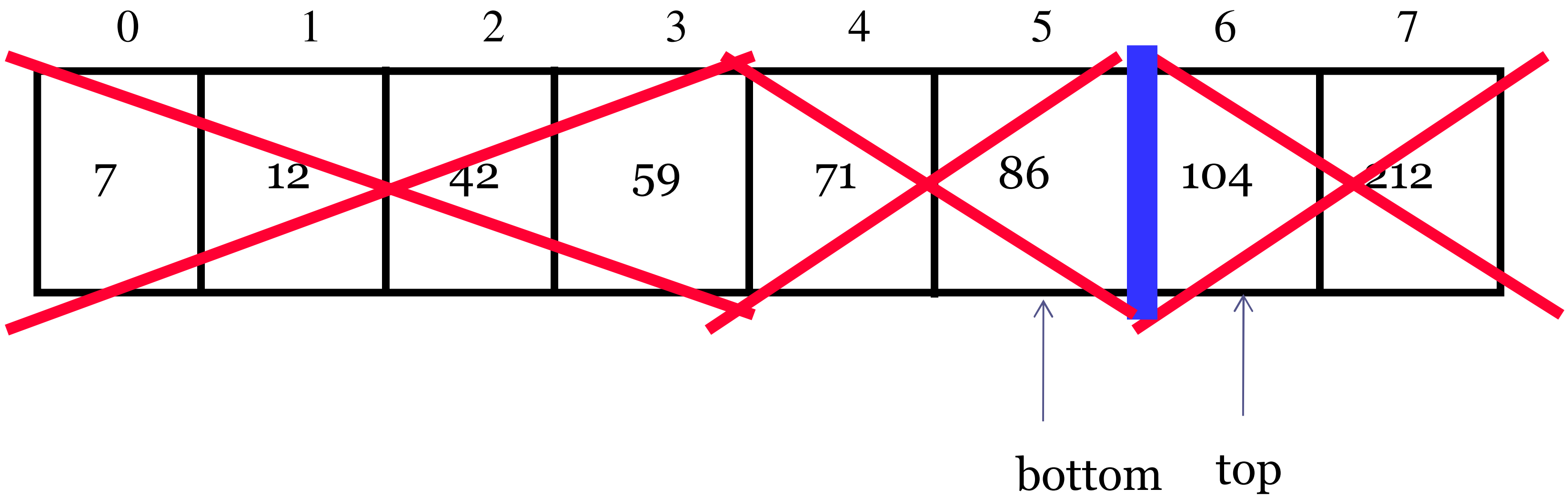
Looking for 89

Binary Search Example



Looking for 89

Binary Search Example



top>bottom: STOP

89 not found – 3 comparisons

$$3 = \text{Log}_2(8)$$

Binary Search Big-O

- **An element can be found by comparing and cutting the work in half.**
  - **We cut work in ½ each time**
  - **How many times can we cut in half?**
  - **Log_2N**

- **Thus binary search is O(Lg N).**

LB

Recall

$$\log_2 N = k \cdot \log_{10} N$$

$$k = 0.30103...$$

So: O(lg N) = O(log N)

Stacks & Queues

- Stacks and Queues are two data structures that allow insertions and deletions operations only at the beginning or the end of the list, not in the middle.
- A stack is a linear structure in which items may be added or removed only at one end.
- A queue is a linear structure in which element may be inserted at one end called the *rear,* and the deleted at the other end called the *front*.

Stacks

- A stack is a list of elements in which an element may be inserted or deleted only at one end, called the ***top of the stack.***

➤ Stacks are also called ***last-in first-out (LIFO)*** lists.

➤ Examples:

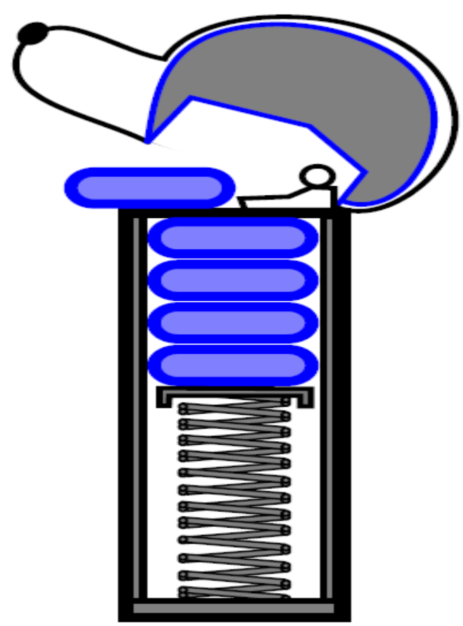
   ➤ a stack of dishes,

   ➤ a stack of coins

   ➤ a stack of folded towels.


Stacks

- Stack has many important applications in computer science.
- The notion of *recursion is fundamental in computer science.*
- *One way of simulating* recursion is by means of stack structure.



Static and Dynamic Stacks

- There are two kinds of stack data structure -

  a) **static**, i.e. they have a **fixed size**, and are *implemented as* **arrays**.

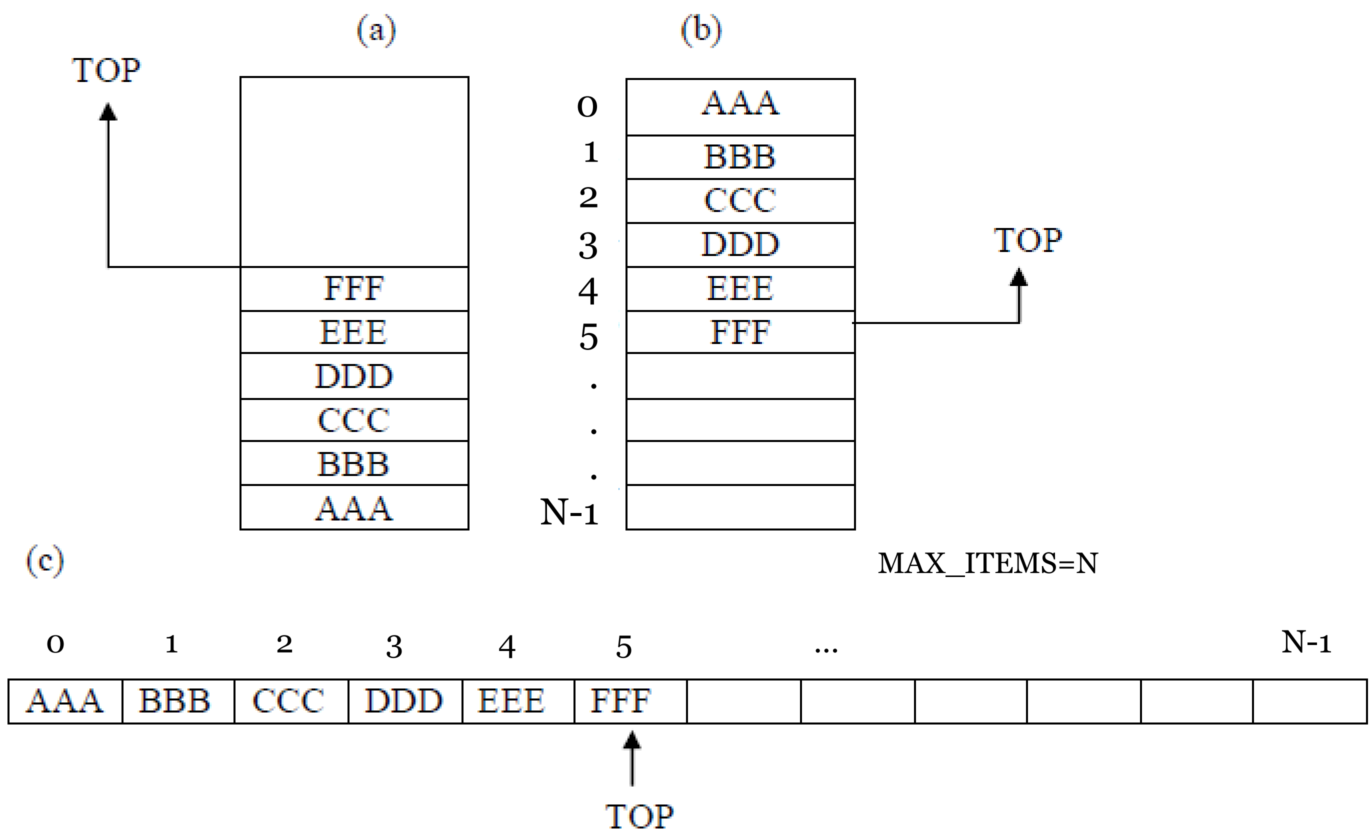
  b) **dynamic**, i.e. they **grow in size** as needed, and *implemented as* **linked lists**.

Stack operations

- Special terminology is used for two basic operation associated with stacks:
  - ***"Push"*** *is the term used to insert an element into a stack.*
  - ***"Pop"*** *is the term used to delete an element from a stack.*
- Apart from these operations, we could perform these operations on stack:
  - Create a stack
  - Check whether a stack is empty,
  - Check whether a stack is full
  - Initialize a stack
  - Read a stack top
  - Print the entire stack.

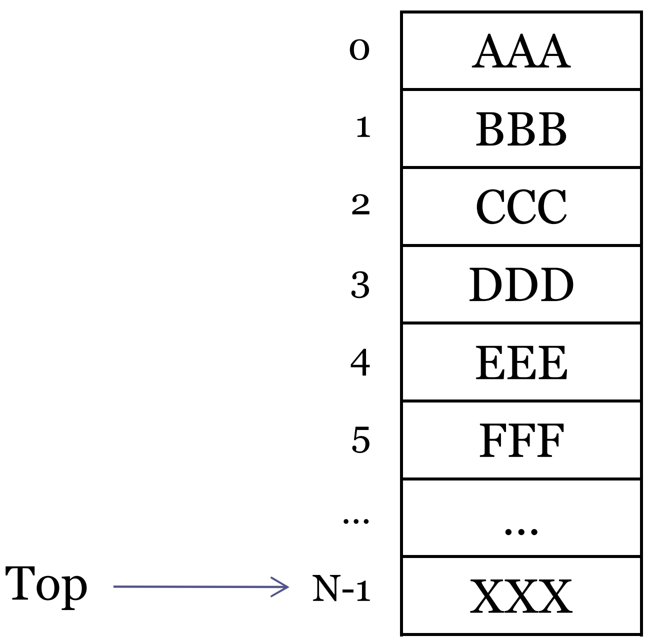
Stack: using array

- Suppose that following 6 elements are pushed, in order, onto an empty stack: AAA, BBB, CCC, DDD, EEE, FFF



Full and Empty Stacks

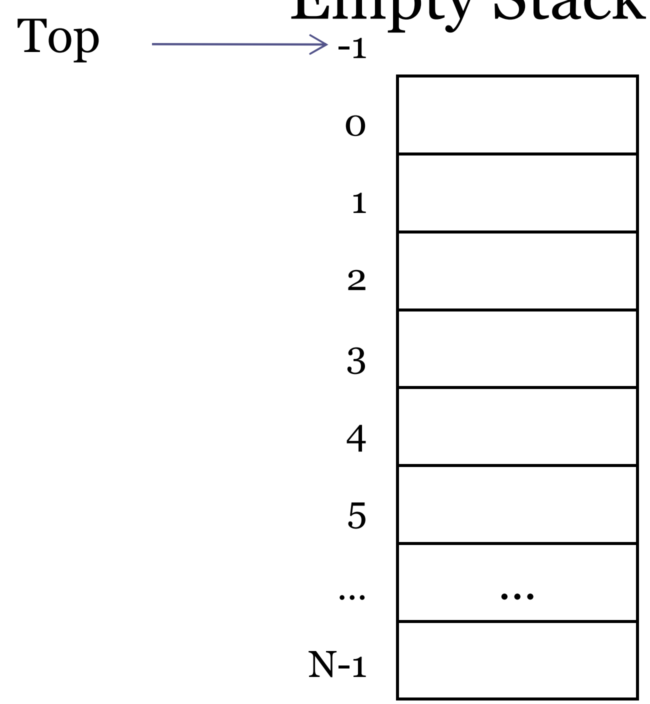
Full Stack



MAX_ITEMS=N

Stack full condition:
Top ==MAX_ITEMS-1

Empty Stack



MAX_ITEMS=N

Stack Empty condition:
Top ==-1

Stack: overflow & underflow

- **Overflow**
  - When we are adding a new element, first, we must test whether there is a free space in the stack for the new item; if not, then we have the condition known as ***overflow***.

- ***Underflow***
  - While removing an element from stack, first test whether there is an element in the stack to be deleted; if not; then we have the condition known as ***underflow*.**

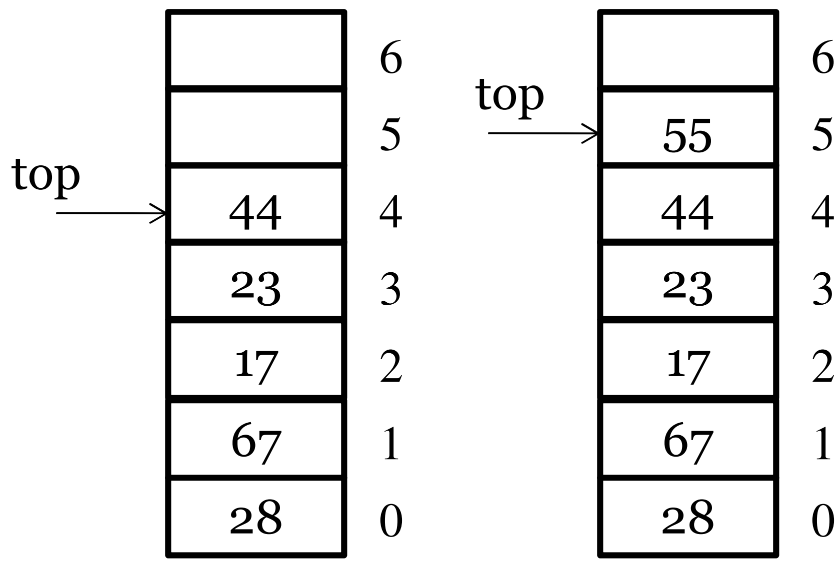
Stack Specification

- Definitions: (provided by the user)
  - ***MAX_ITEMS***: Max number of items that might be on the stack
  - ***itemType*****:** Data type of the items on the stack
- Operations
  - **Initialize( )**
  - **Boolean isEmpty( )**
  - **Boolean isFull( )**
  - **Push (itemType newItem)**
  - **itemType Pop ( )**

Push (ItemType newItem)

- *Function*: Adds newItem to the top of the stack.
- *Preconditions*: Stack has been initialized and is not full.
- *Postconditions*: newItem is at the top of the stack.

```
Push (newItem)
{
        IF (isFull()) THEN
                print "STACK
OVERFLOW";
        ELSE
        {
                top=top+1;
                Stack[top]=newItem;
        }
}
```



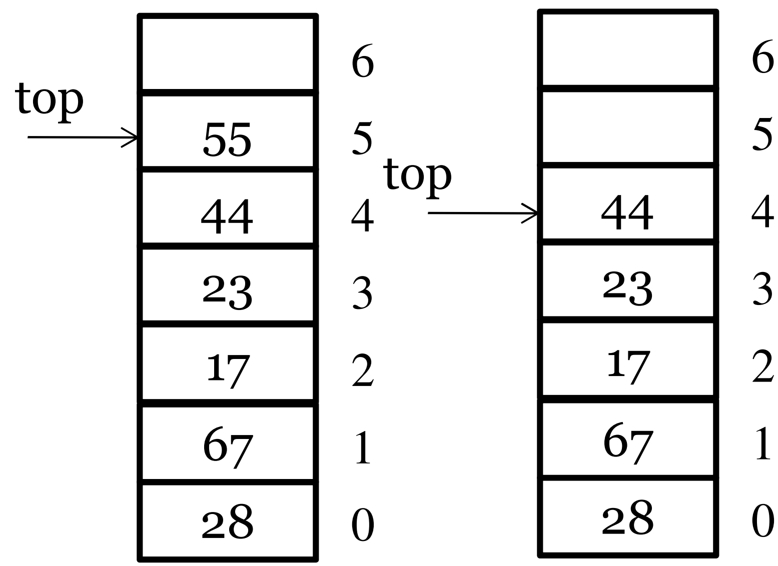
SAMSUNG

DTU Delhi Technological University

itemType Pop ( )

- *Function*: Removes topItem from stack and returns it.
- *Preconditions*: Stack has been initialized and is not empty.
- *Postconditions*: Top element has been removed from stack and returned.

```
itemType Pop( )
{
        IF (isEmpty()) THN
                print " STACK UNDERFLOW";
        ELSE
        {
                topItem=Stack[top];
                top=top-1;
                return(topItem);
        }
}
```



Ceate a stack

```
itemType  Stack[MAX_ITEMS]
int top;
```

Initialize a stack

```
Initialize()
{
        top = -1;
}
```

isEmpty()

```
Boolean isEmpty()
{
        IF (top == -1) THEN
                return(TRUE);
        ELSE
                return(FALSE);
}
```

isFull()

```
Boolean isFull()
{
        IF (top == MAX_ITEMS-1) THEN
                return(TRUE);
        ELSE
                return(FALSE);
}
```

Stack class

```
class Stack {
public:
        Stack(int size = 10);                          // constructor
        ~Stack() { delete [] values; }    // destructor
        bool IsEmpty() { return top == -1; }
        bool IsFull() { return top == maxTop; }
        double Top();
        void Push(const double x);
        double Pop();
        void DisplayStack();
private:
        int maxTop;            // max stack size = size - 1
        int top;               // current top of stack
        double* values;        // element array
};
```

Create Stack

- The constructor of `Stack`
  - Allocate a stack array of `size`. By default, `size = 10`.
  - When the stack is full, `top` will have its maximum value, i.e. `size - 1`.
  - Initially `top` is set to -1. It means the stack is empty.

```
Stack::Stack(int size=10) {
        maxTop =      size - 1;
        values =      new double[size];
        top           =      -1;
}
```

Although the constructor dynamically allocates the stack array, the stack is still static. The size is fixed after the initialization.

Push Stack

- `void Push(const double x);`
  - Push an element onto the stack
  - If the stack is full, print the error information.
  - Note `top` always represents the index of the top element. After pushing an element, increment `top`.

```
void Stack::Push(const double x) {
        if (IsFull())
                cout << "Error: the stack is full." << endl;
        else
                values[++top] = x;
}
```

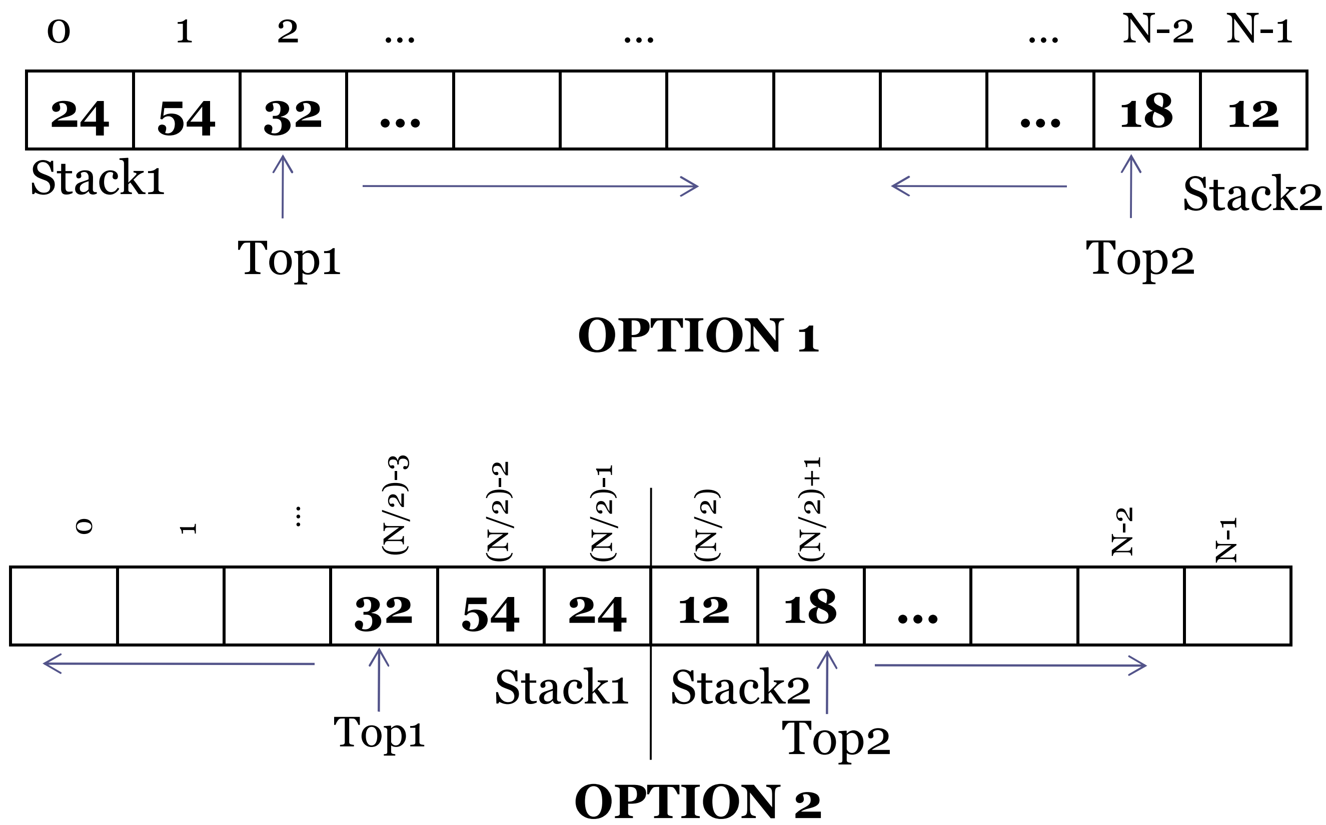
Pop Stack

- `double Pop()`
  - Pop and return the element at the top of the stack
  - If the stack is empty, print the error information. (In this case, the return value is useless.)
  - Don't forgot to decrement `top`

```
double Stack::Pop() {
	if (IsEmpty()) {
		cout << "Error: the stack is empty." << endl;
		return -1;
	}
	else {
		return values[top--];
	}
}
```

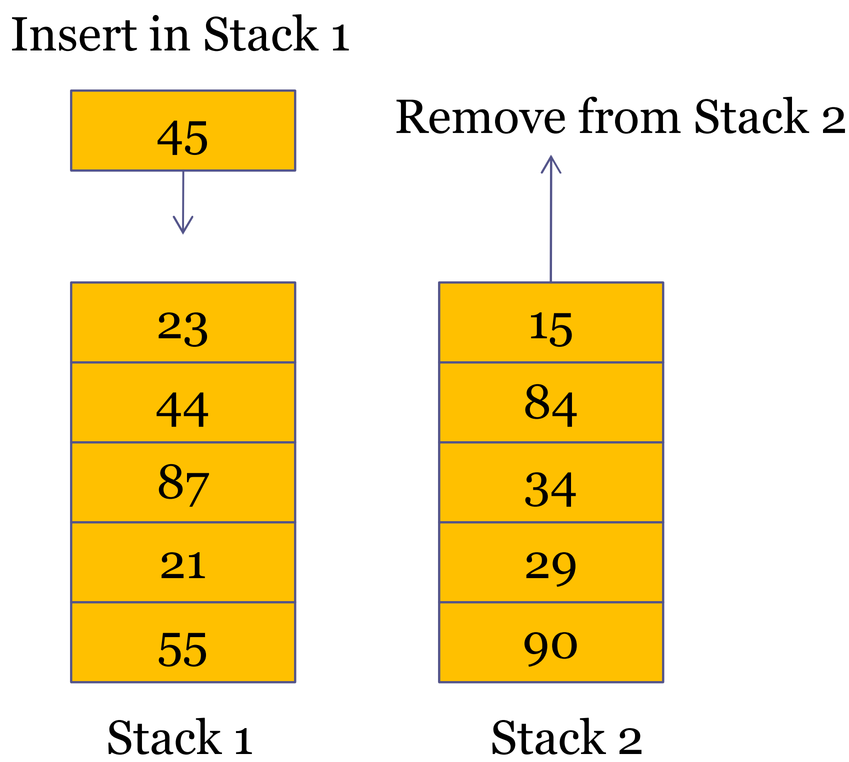
Stacks: Sharing space

- Two stacks sharing space of single array



Which option is better ?

Queue using two stacks



What if Stack2 is empty and request for dequeue?
What if Stack1 is full and request for enqueue?

Queues

Queue Basics

- A queue is a sequence of data elements
- In the sequence
  - Items can be removed only at the front
  - Items can be added only at the other end, the **rear**

Samsung DTU Delhi Technological University

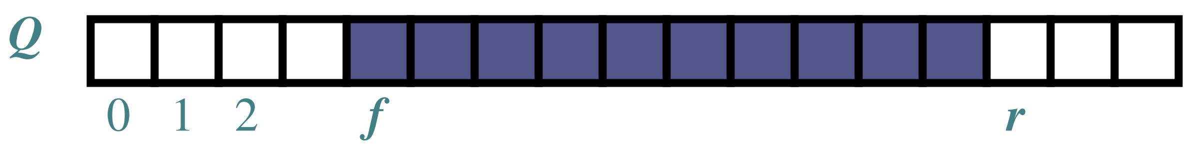
Array-based Queue

- Use an array of size N in a circular fashion
- Two variables keep track of the front and rear

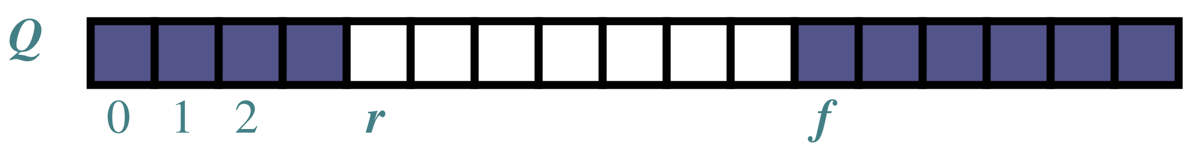
  f index of the front element

  r index immediately past the rear element
- Array location r is kept empty

normal configuration

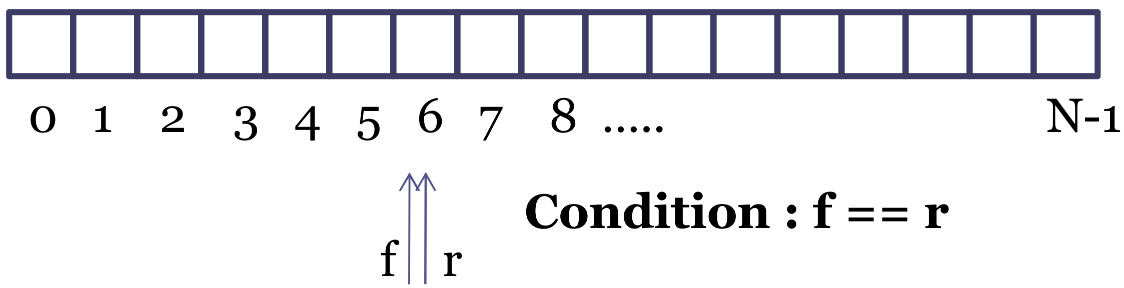


wrapped-around configuration



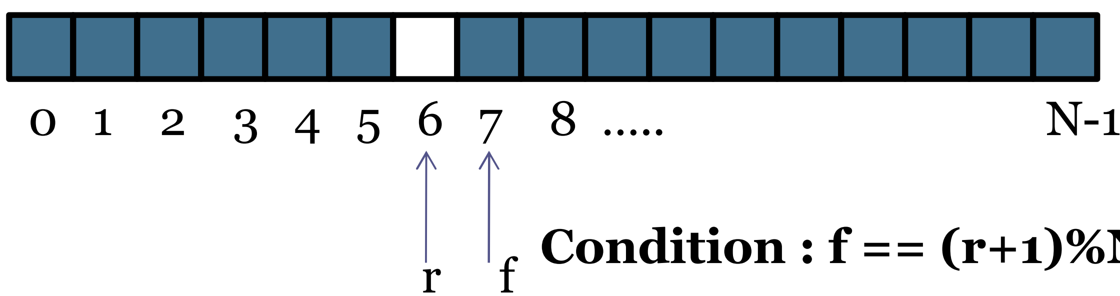
Empty and Full Queue (wrapper-around configuration)

Empty Queue



Condition : f == r

Full Queue



Condition : f == (r+1)%N

Queue operations

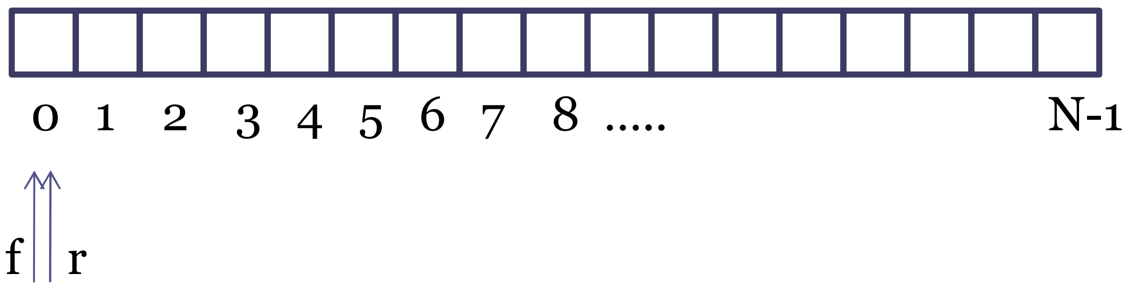
- Special terminology is used for two basic operation associated with queues:
  - **"enqueue"** *is the term used to insert an element at the end of the queue.*
  - **“dequeue"** *is the term used to remove an element at the front of the queue.*
- Apart from these operations, we could perform these operations on queue:
  - Create a queue
  - Check whether a queue is empty,
  - Check whether a queue is full
  - Initialize a queue
  - Read front element of the queue
  - Print the entire queue.

SAMSUNG

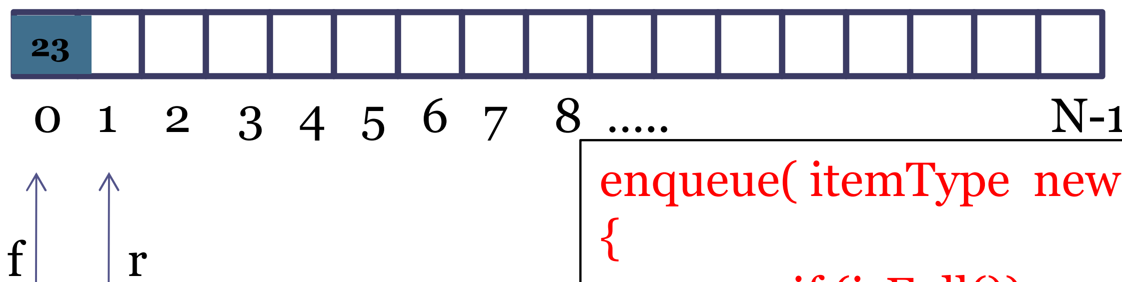
DTU
Delhi Technological University

Queue: enqueue : example

Empty Queue



After adding 23 : enqueue(23)




```
enqueue( itemType  newItem)
{
        if (isFull())
                print "queue is Full";
        else
        {
                Queue[r]=newItem;
                r=(r+1) mod MAX_ITEMS;
        }
}
```

Queue: enqueue: example…

After adding 44 : enqueue(44)

23	44															

0 1 2 3 4 5 6 7 8 ..... N-1

f r

After adding 12, 35, 27, 65, 89

23	44	12	35	27	65	89										

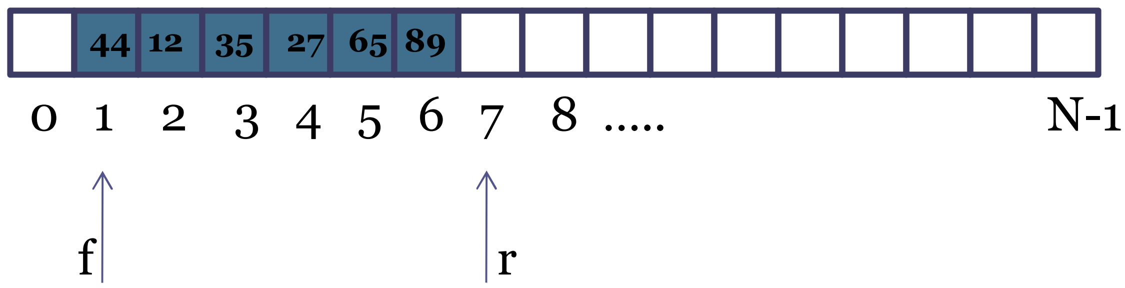
0 1 2 3 4 5 6 7 8 ..... N-1

f r

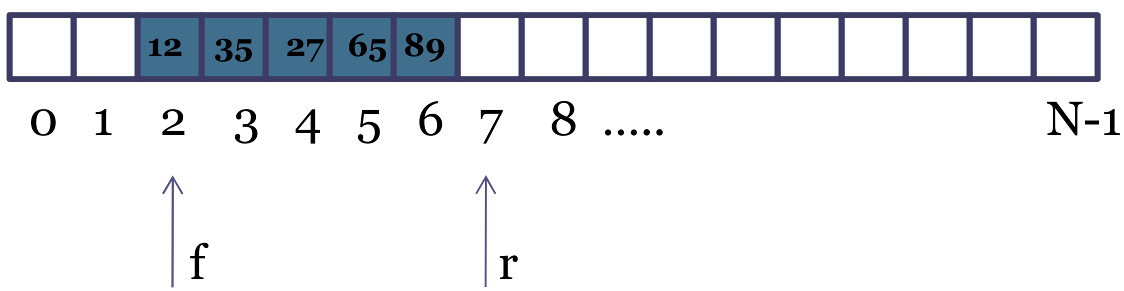
```
enqueue( itemType  newItem)
{
        if (isFull())
                print "queue is Full";
        else
        {
                Queue[r]=newItem;
                r=(r+1) mod MAX_ITEMS;
        }
}
```

Queue: dequeue: example…

After removing one element (23) : dequeue()



After removing next element (44) : dequeue()



Queue: dequeue operation

```
itemType  dequeue()
{
        IF (isEmpty()) THEN
                print "queue is Empty";
        ELSE
        {
                frontItem=Queue[f];
                f=(f+1) mod MAX_ITEMS;
                return(frontItem);
        }
}
```

Queue creation

```
itemType  Queue[MAX_ITEMS];
int  f, r;
```

Queue initialization

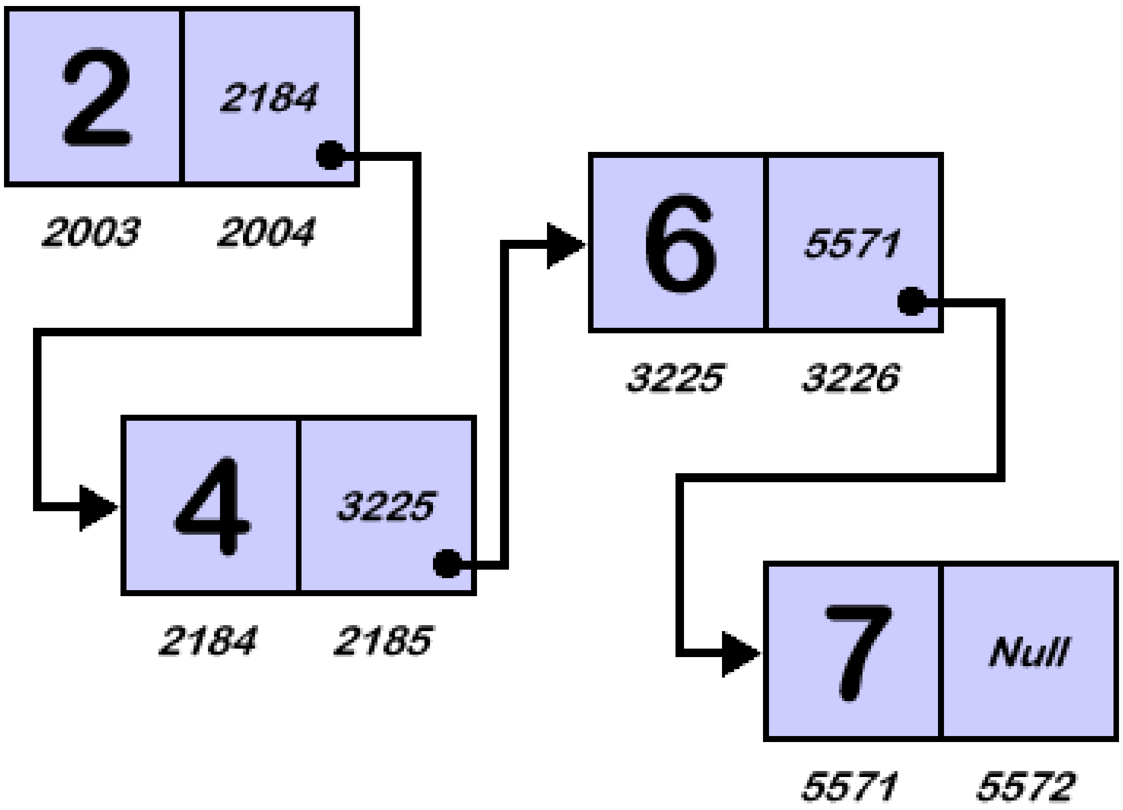
```
f = 0;
r = 0;
```

Queue: isEmpty(), isFull()

```
Boolean isEmpty()
{
        IF (f == r) THEN
                return(TRUE);
        ELSE
                return(FALSE);
}
```

```
Boolean isFull()
{
        IF (f == (r + 1) mod MAX_ITEMS) THEN
                return(TRUE);
        ELSE
                return(FALSE);
}
```

Linked Lists



Linked list

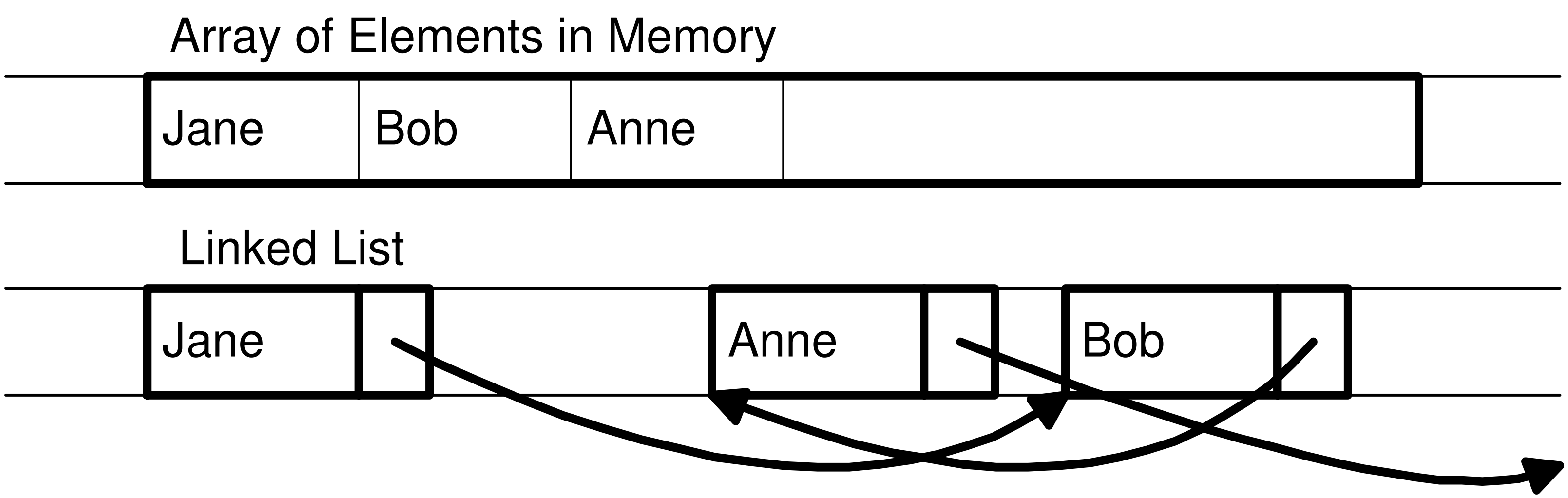
- Alternate approach to maintaining an array of elements
- Rather than allocating one large group of elements, allocate elements as needed
- Q: how do we know what is part of the array?
  - A: have the elements keep track of each other
  - use pointers to connect the elements together as a *LIST* of things
- Allows efficient insertion and removal, sequential access

Limitations of array

- An array has a limited number of elements
  - routines inserting a new value have to check that there is room
- Can partially solve this problem by reallocating the array as needed (how much memory to add?)
  - adding one element at a time could be costly
  - one approach - double the current size of the array
- A better approach: use a *Linked List*

Dynamically Allocating Space for Elements

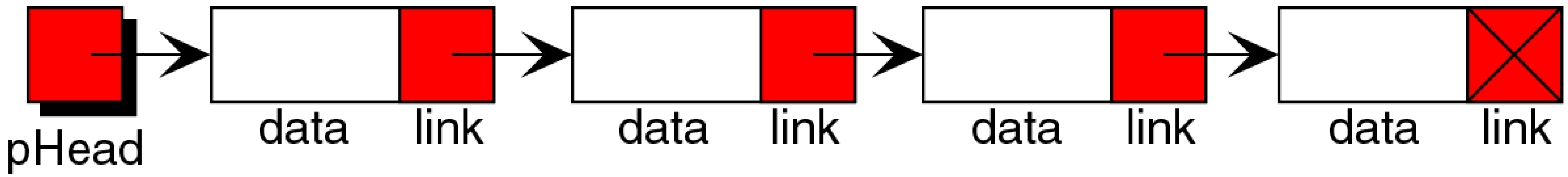
- Allocate elements one at a time as needed, have each element keep track of the *next* element
- Result is referred to as linked list of elements, track next element with a pointer



Linked List Notes

- Need way to indicate end of list (NULL pointer)
- Need to know where list starts (first element)
- Each element needs pointer to next element (its link)
- Need way to allocate new element (use malloc)
- Need way to return element not needed any more (use free)
- Divide element into data and pointer

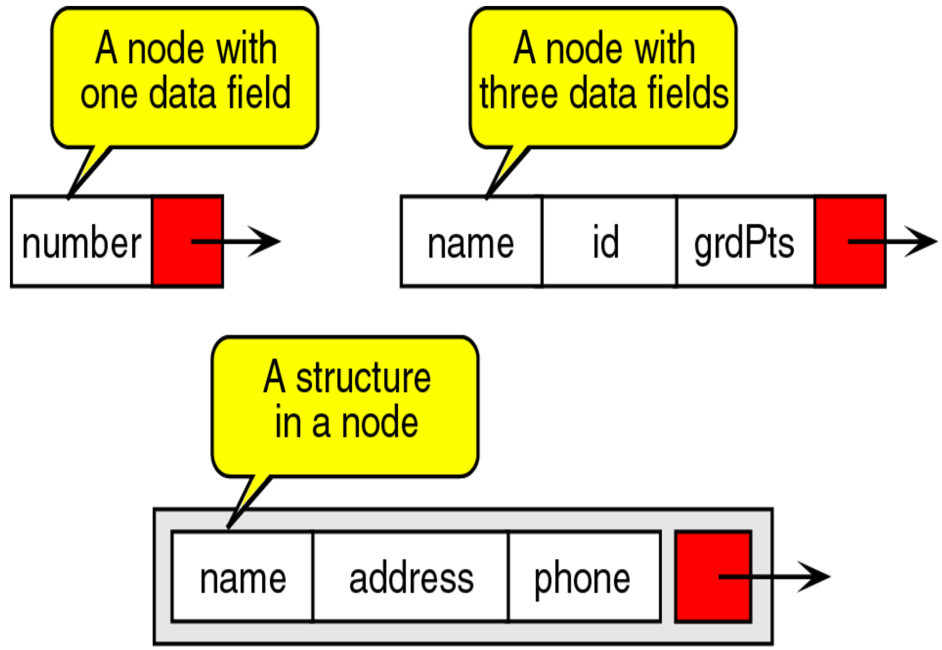
Linked list concept



- Here we see a basic linked list.
- There are 4 elements in the list, each one with a data portion and a link portion.
- **pHead** is a pointer to the head of the list. Typically, the name given to this pointer is the name of the list.
- Note the last element of the list. The **X** in the link portion denotes a NULL pointer (i.e., the end of the list).

Nodes

- The elements in a linked list are traditionally called nodes.
- A node in a linked list is a structure that has at least 2 fields: one contains the data, the other contains the address of the next element in the list.
- A node can contain data of any type, including objects of other classes.



Nodes

- The nodes that make up a linked list are ***self-referential*** structures.
- A self-referential structure is one in which each instance of the structure contains a pointer to another instance of the same structural type.

Linked list concept

- Data is stored in a linked list dynamically – each node is created as required.
- Nodes of linked lists are not necessarily stored contiguously in memory (as in an array).
- Although lists of data can be stored in arrays, linked lists provide several advantages.

Linked list concept

Advantage 1: Dynamic

- A linked list is appropriate when the number of data elements to be stored in the list is unknown.
- Because linked lists are dynamic, their size can grow or shrink to accommodate the actual number of elements in the list.

Linked list concept

- The size of a "conventional" C++ array, however, cannot be altered, because the array size is fixed at compile time.
- Also, arrays can become full (i.e., all elements of the array are occupied).
- A linked list is full only when the computer runs out of memory in which to store nodes.

Linked list concept

Advantage 2: Easy Insertions and Deletions

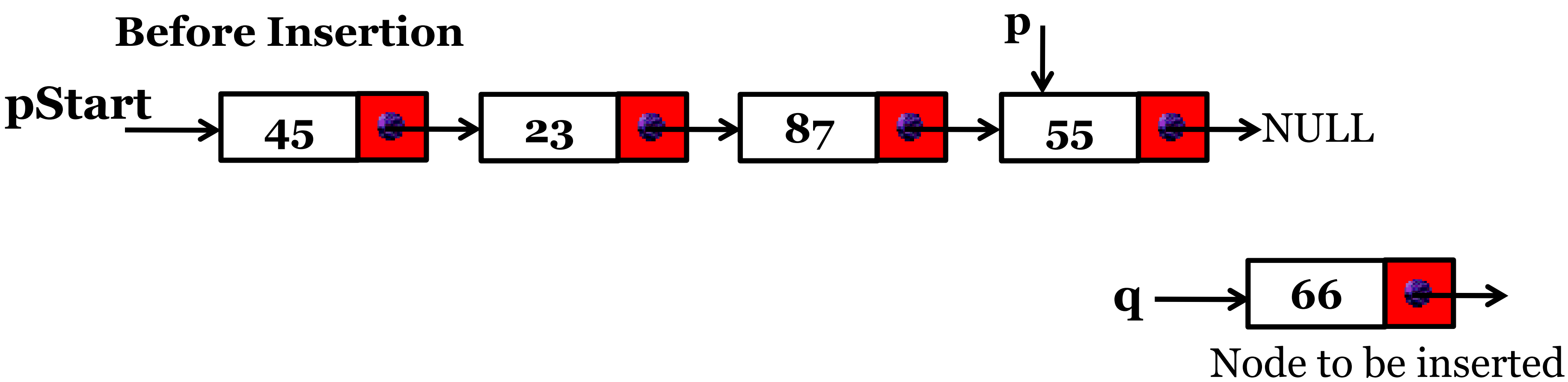
- Although arrays are easy to implement and use, they can be quite inefficient when sequenced data needs to be inserted or deleted.
- With arrays, it is more difficult to rearrange data.
- However, the linked list structure allows us to easily insert and delete items from a list.

Linked list concept

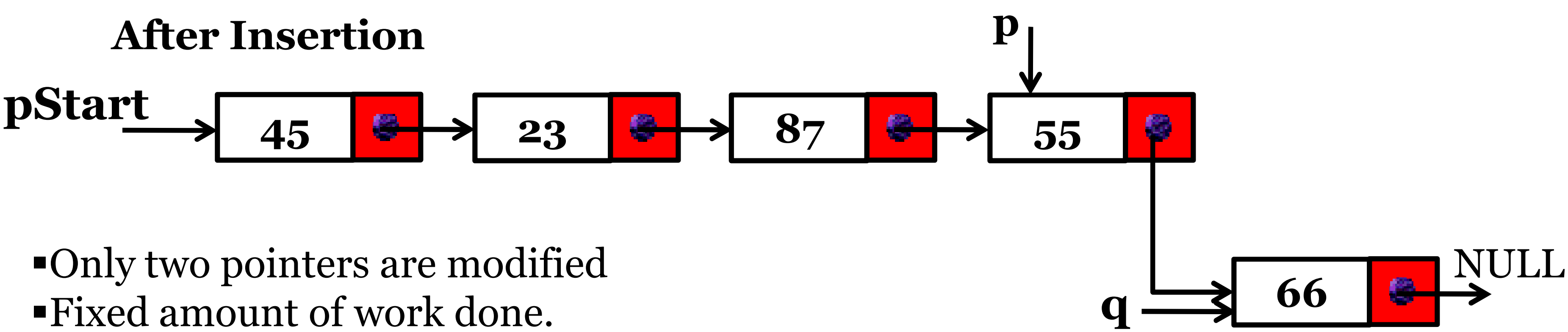
- Unfortunately, linked lists are not without their drawbacks.
- For example, we can perform efficient searches on arrays (e.g., binary search), but this is not practical with a linked list.

Singly Linked list: Insertion at end

Before Insertion



After Insertion



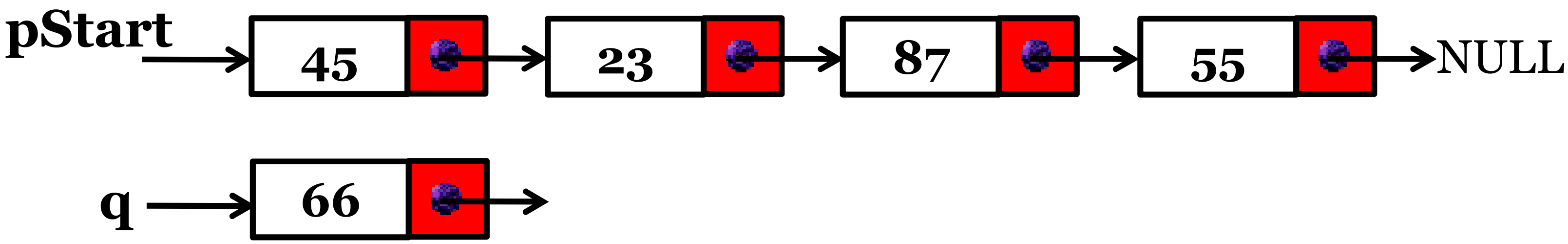
- Only two pointers are modified
- Fixed amount of work done.
- O(1) complexity

Operations

```
p->next = q;
q->next = NULL;
```

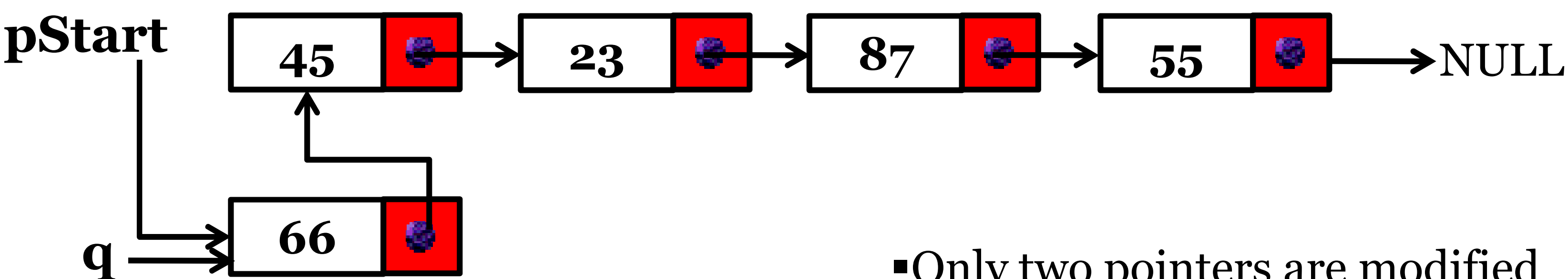
Linked list: Insertion at start

Before Insertion



Node to be inserted

After Insertion

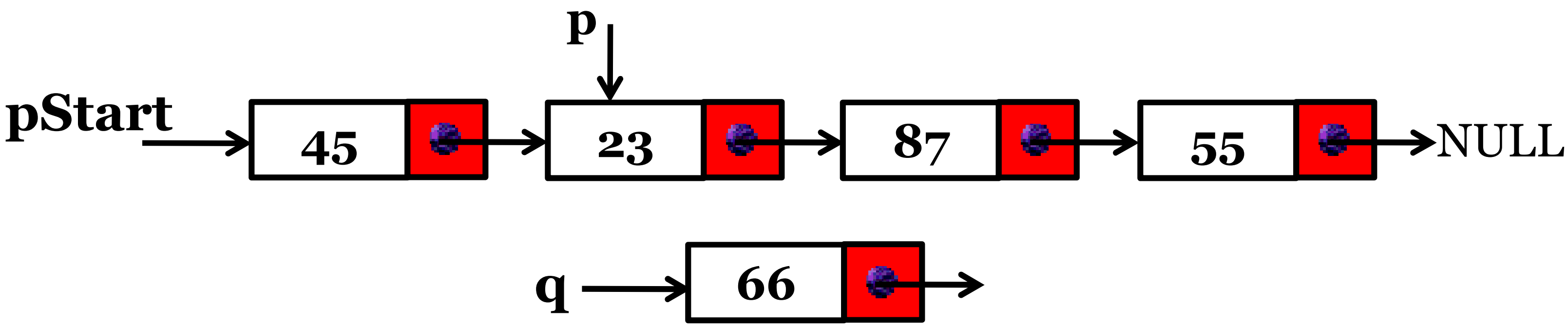



```
q->next = pStart;
pStart = q;
```

- Only two pointers are modified
- Fixed amount of work done.
- O(1) complexity

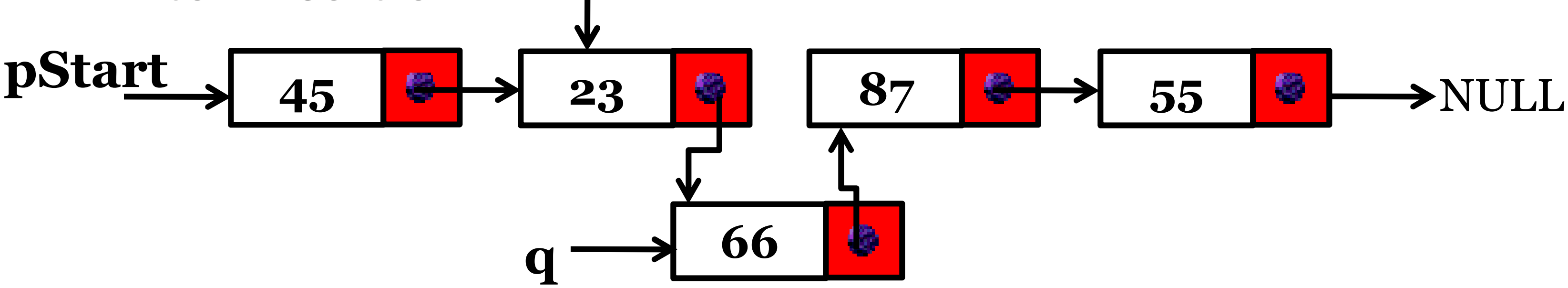
Linked list: Insertion in middle

Before Insertion: request to insert after node pointed by p



Node to be inserted

After Insertion




```
q->next = p->next;
p->next = q;
```

- Only two pointers are modified
- Fixed amount of work done.
- O(1) complexity

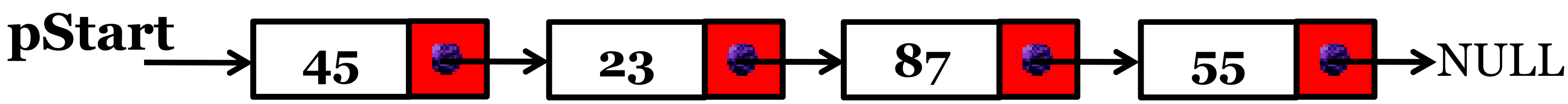
Node: declaration in C

rollno	name	marks	next

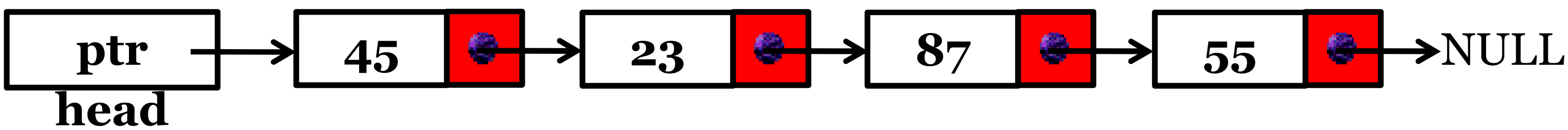
```
typedef struct node        //defining node structure
{
          int          rollno;
          char         name[30];
          int          marks;
          struct node *next;
} ;
struct node *pStart, *p, * q ; // creating pointers variacles
q = (struct node*) (malloc(sizeof(struct node)));
                               // creating a new node pointed by q
```

Concept of head node

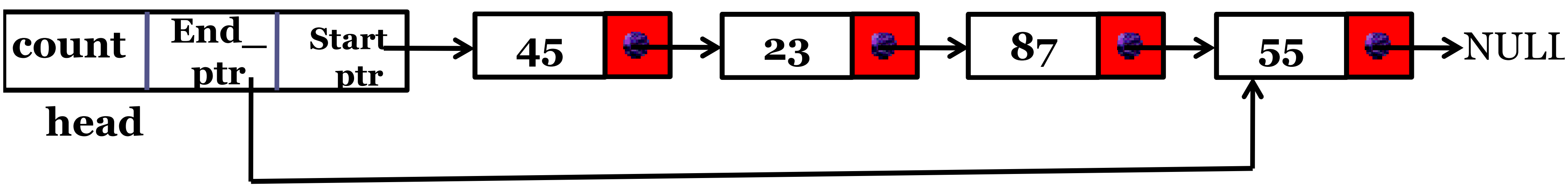
Linked list without head node



Linked list with head node

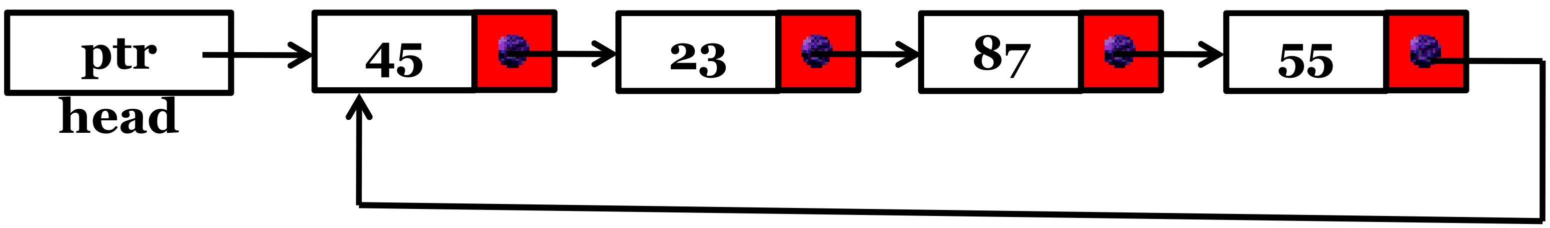


Linked list with head node storing number of nodes in linked list and pointers to first and last node

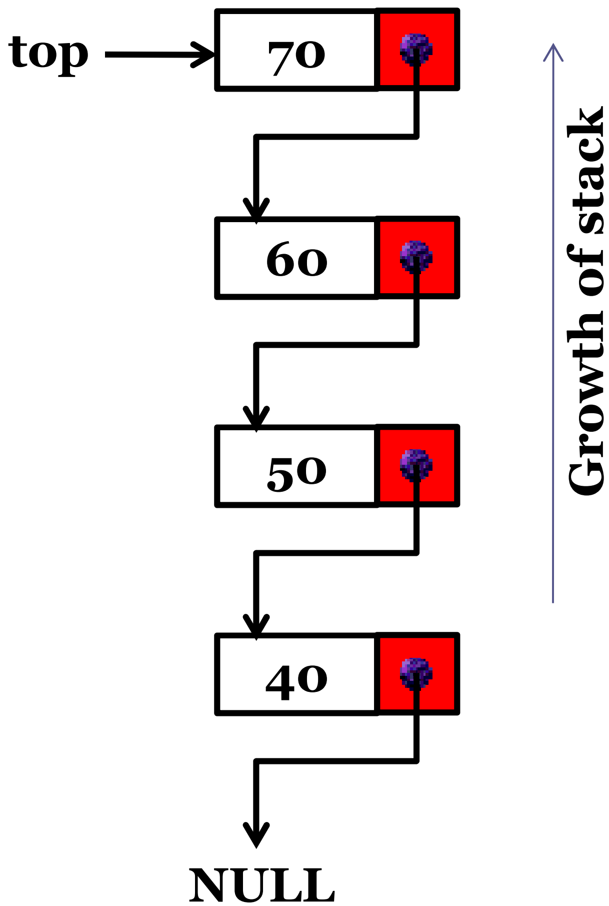


Circular linked list

- Next pointer of last node points to first node of the list



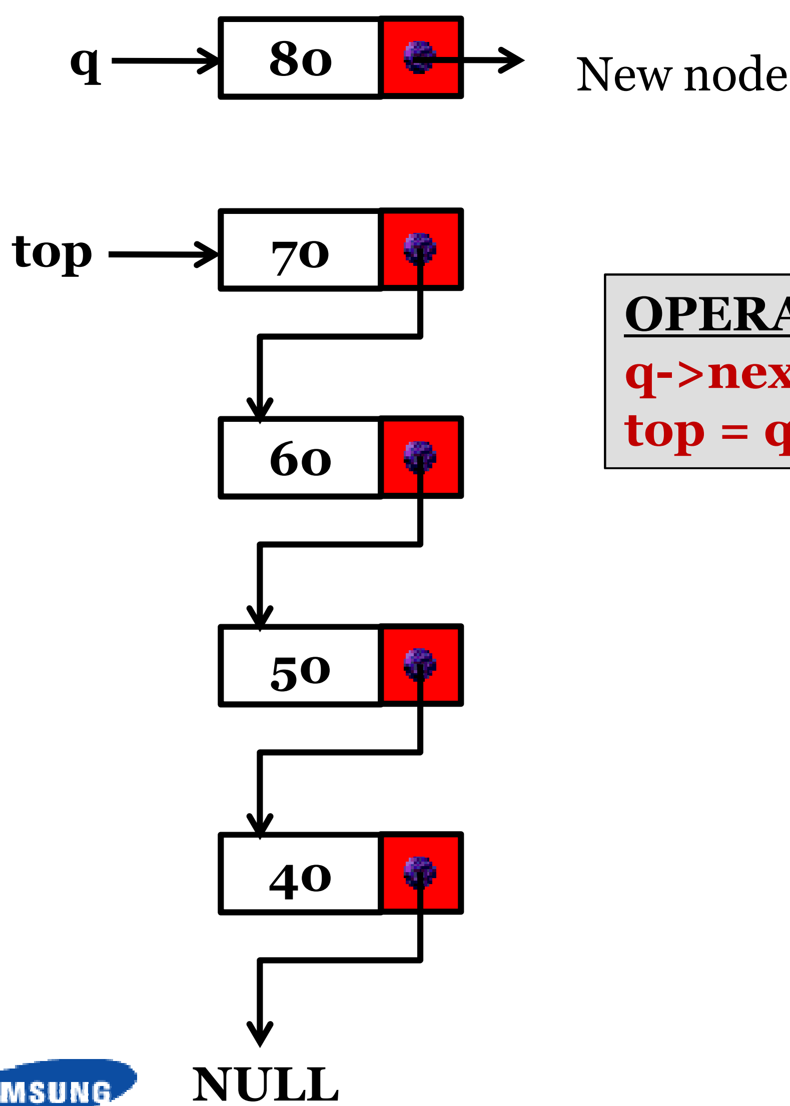
Stack using linked list



top ⟶ NULL

Empty Stack

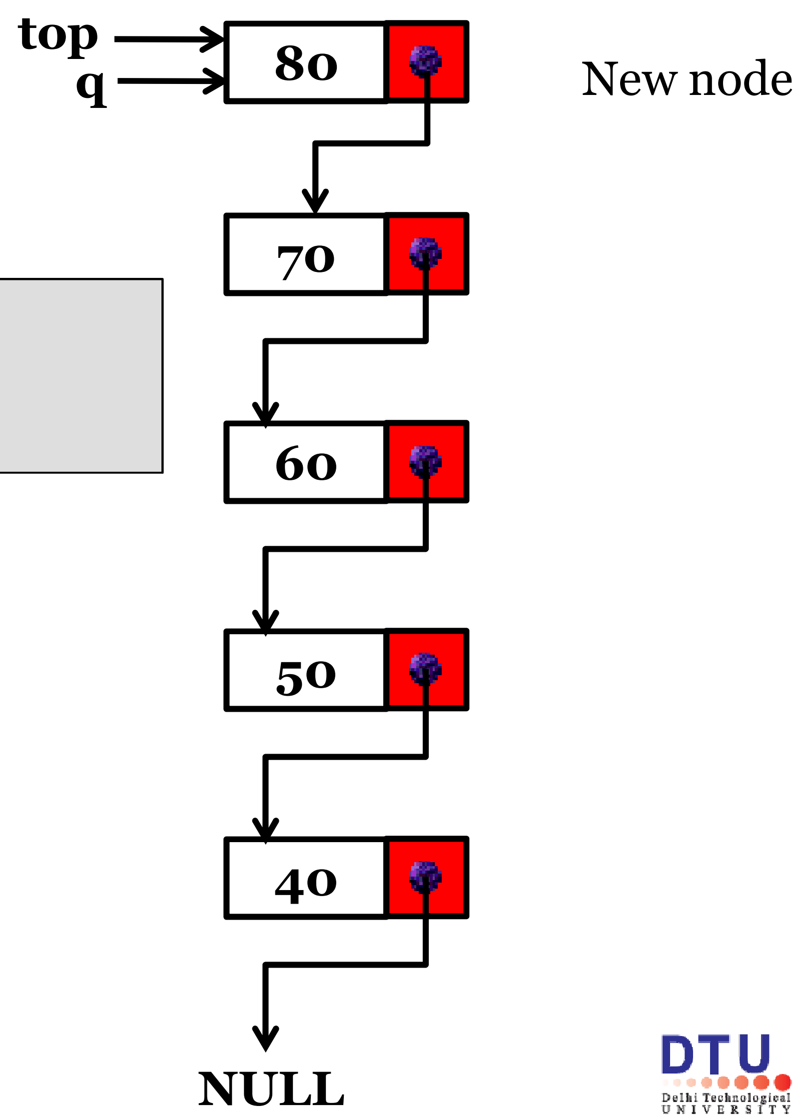
Stack using linked list : PUSH operation



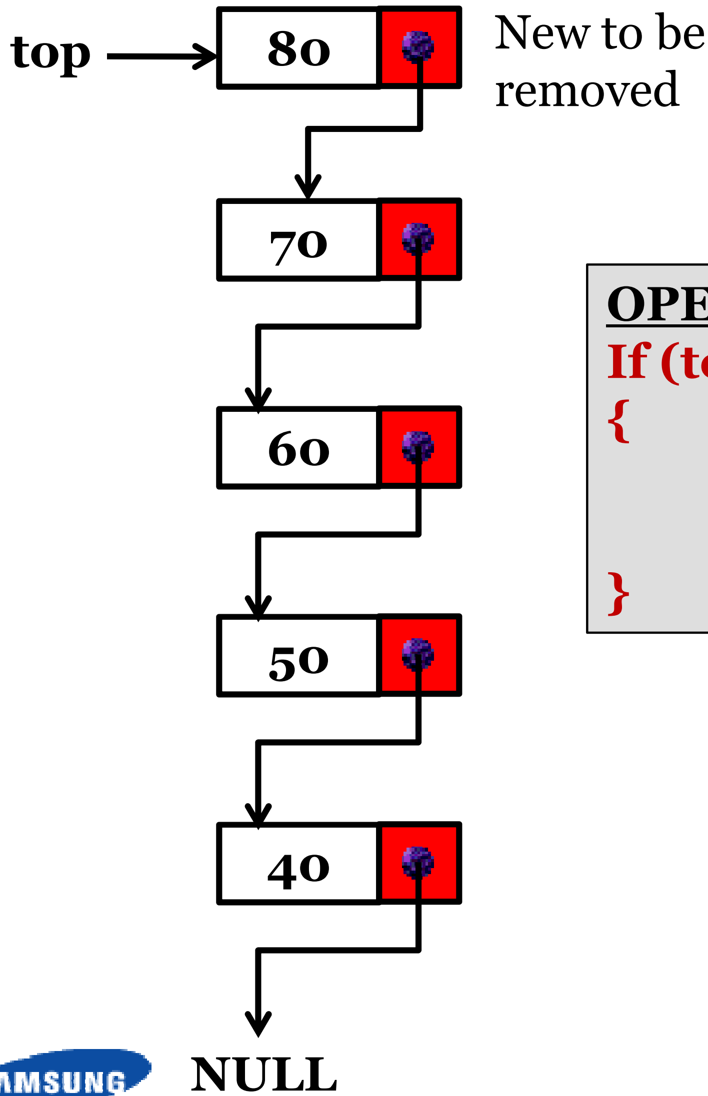
OPERATIONS

q->next = top;

top = q;

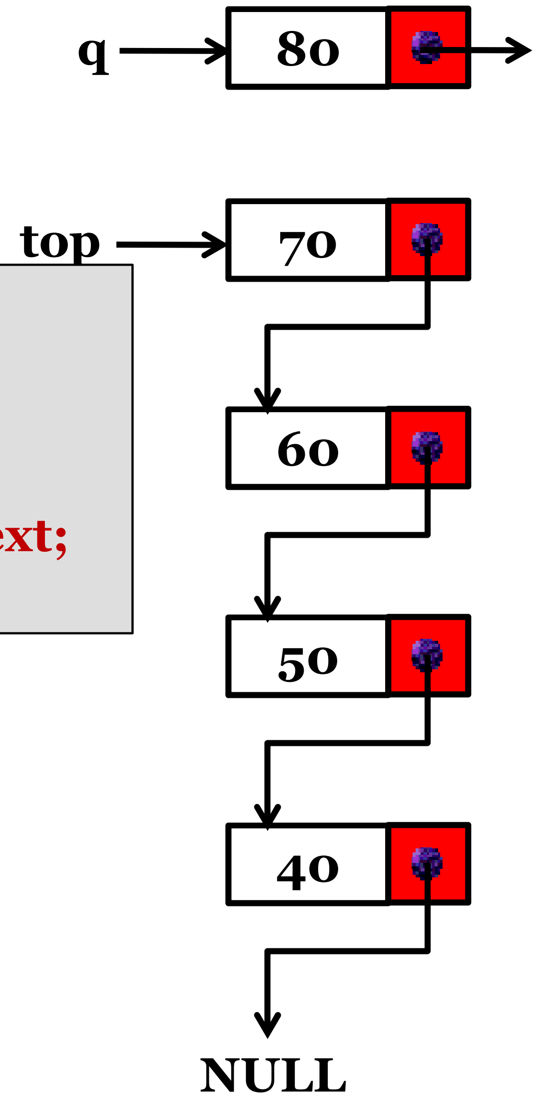


Stack using linked list : POP operation

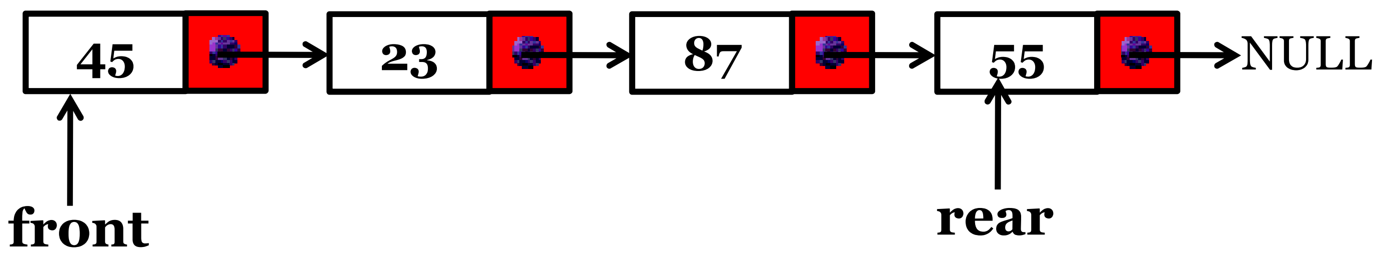


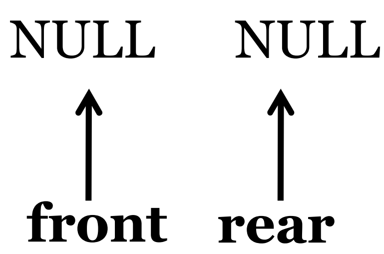
OPERATIONS

```
If (top != NULL)
{
        q = top;
        top = top->next;
}
```



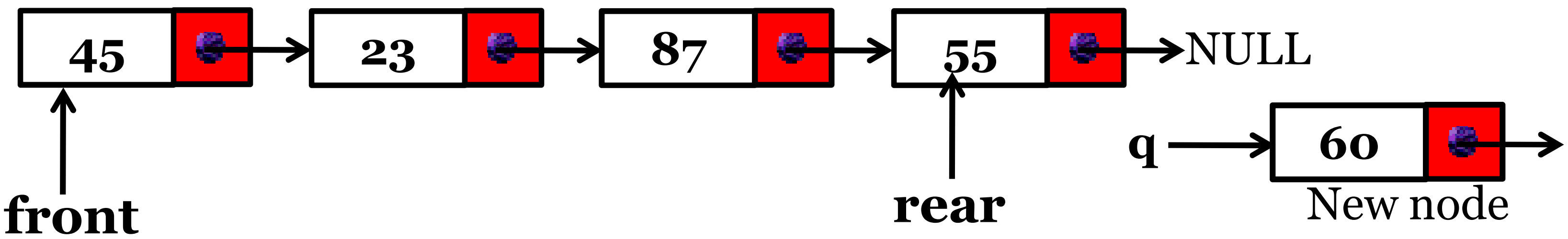
Queue using linked list

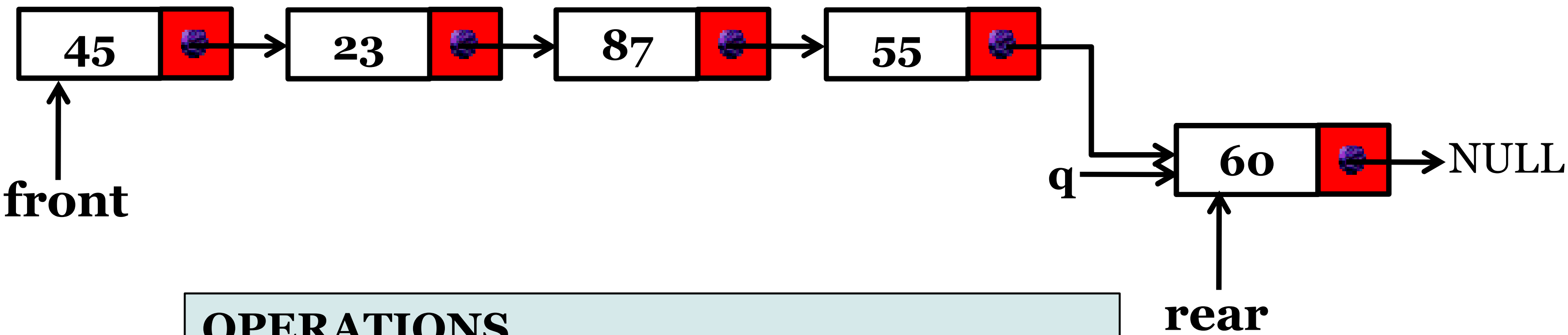




Empty queue:
front = rear = NULL

Queue using linked list: enqueue operation

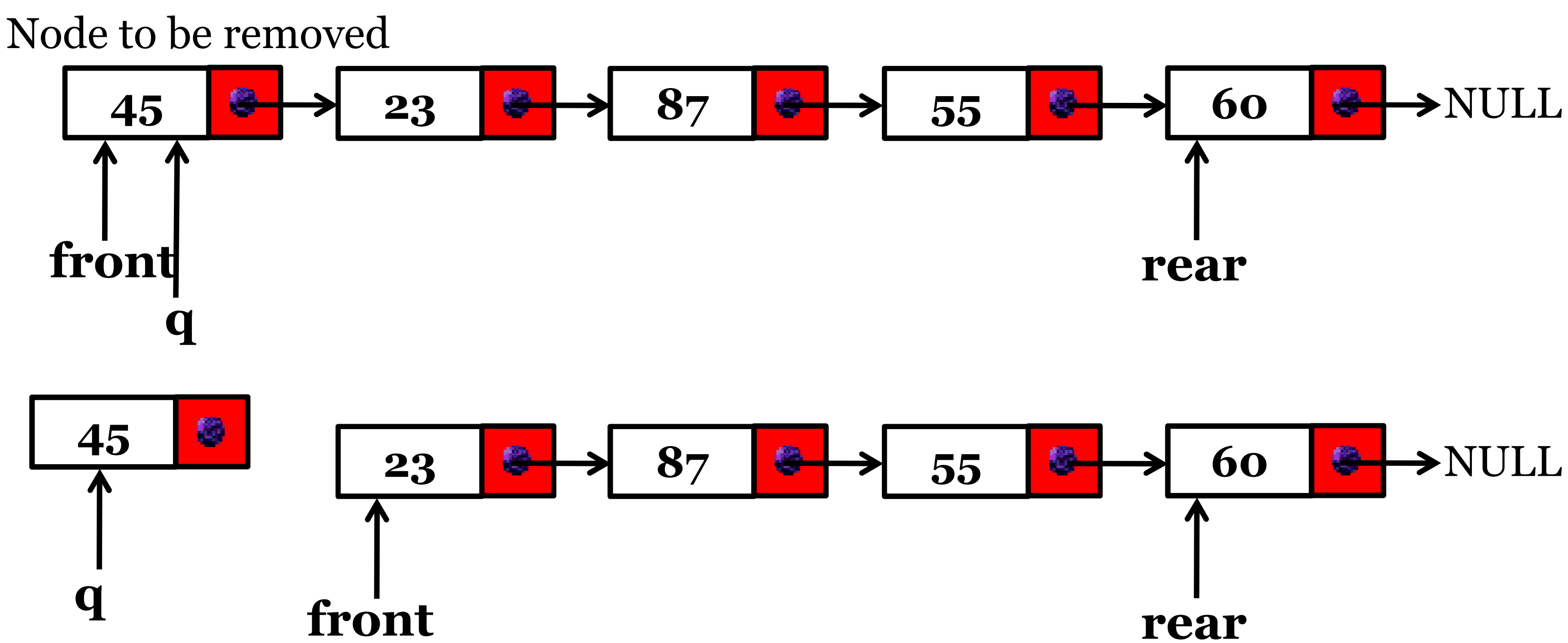




OPERATIONS

```
If  (rear  != NULL)
{
            rear ->next = q;
            rear = q;
}
else        // empty queue
            rear = q; front = q;
```

Queue using linked list: dequeue operation



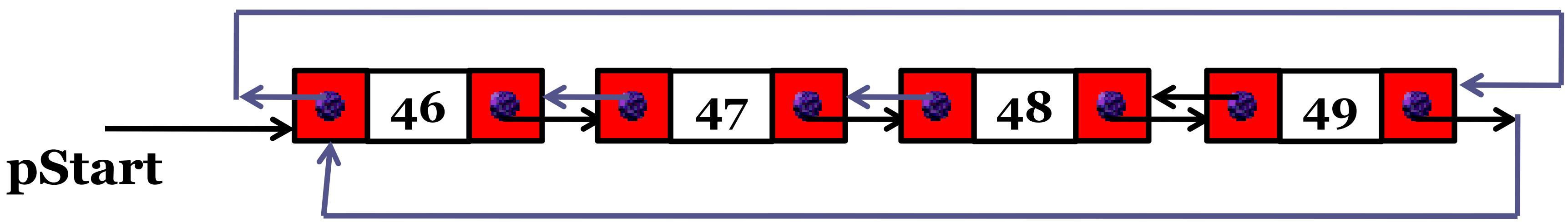
OPERATIONS

```
q = front;
If  (front  != NULL)
{
        front = front ->next;
        if (front == NULL) rear = NULL;
}
return (q);
```

Doubly Linked Lists (DLLs)

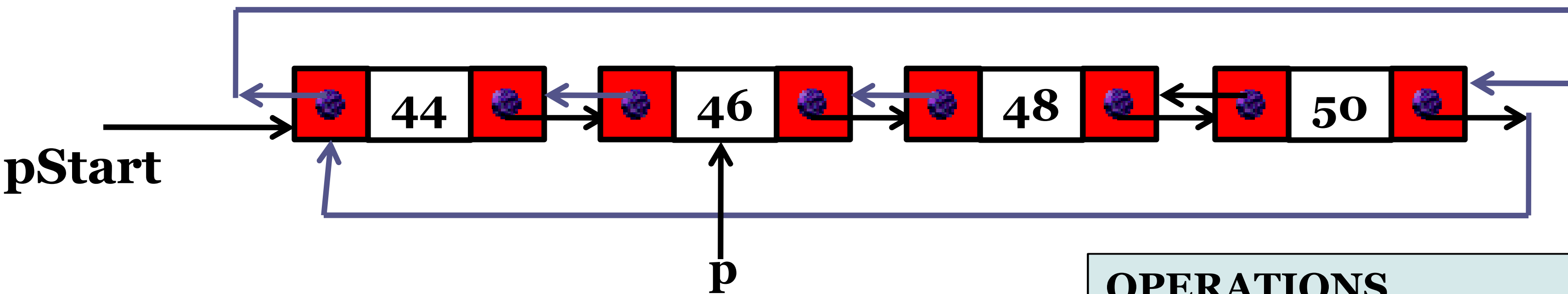
Node structure

Circular Doubly Linked List

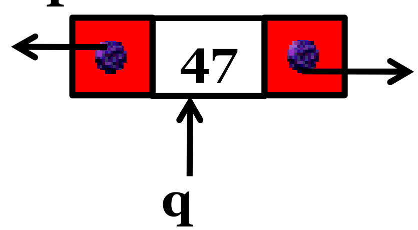


Node structure

Doubly Linked List: Insertion

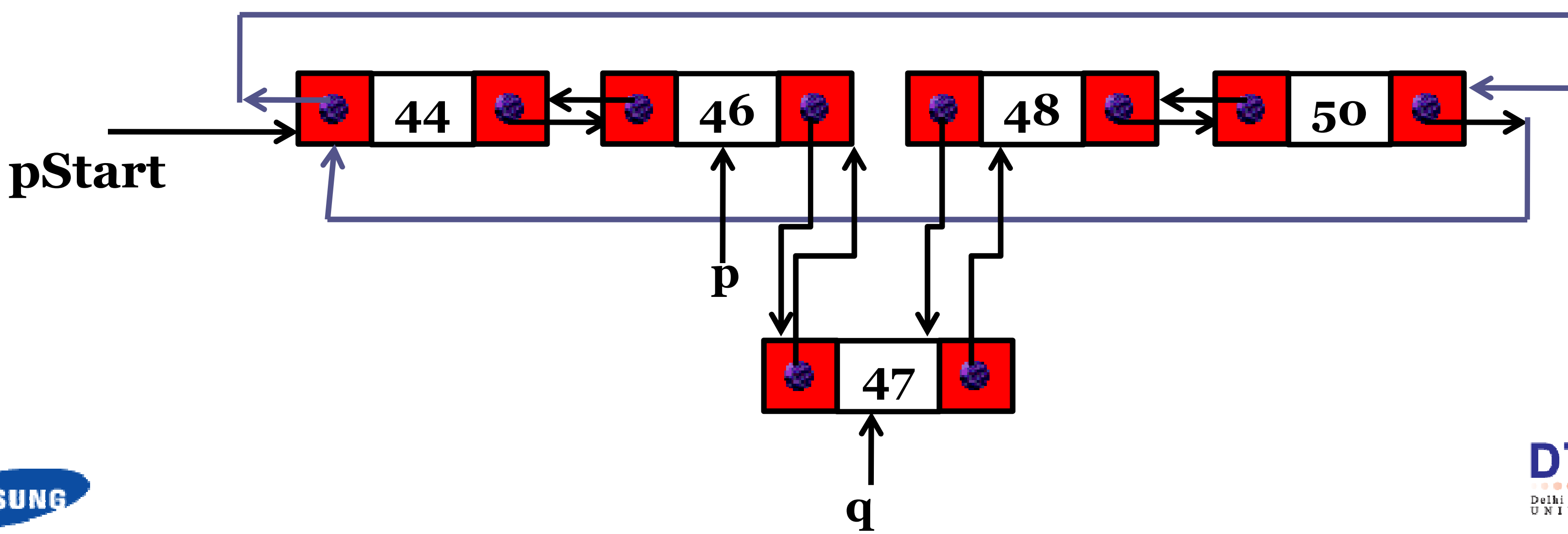


Insert new node q after node p

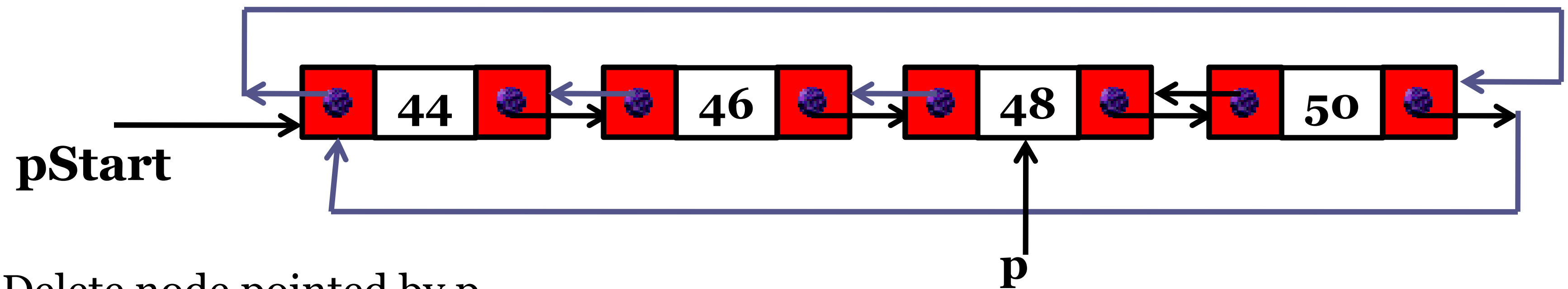


OPERATIONS

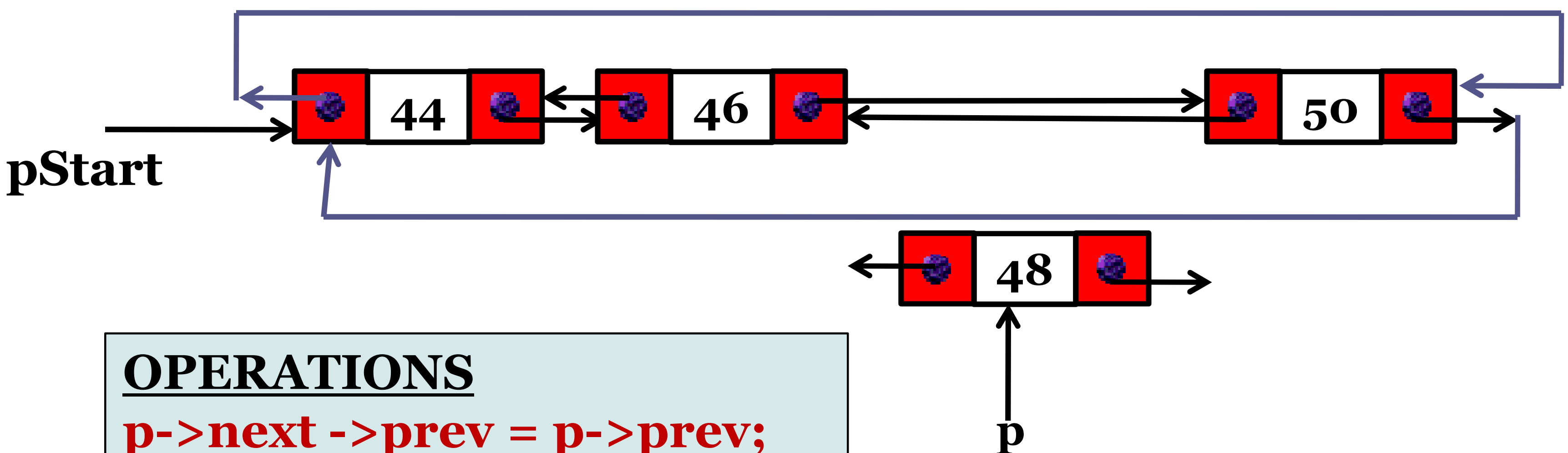
q->next = p->next;
q->prev = p;
p->next->prev = q;
p->next = q;



Doubly Linked List: Deletion



Delete node pointed by p



<u>OPERATIONS</u>

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

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

free(p); //release memory

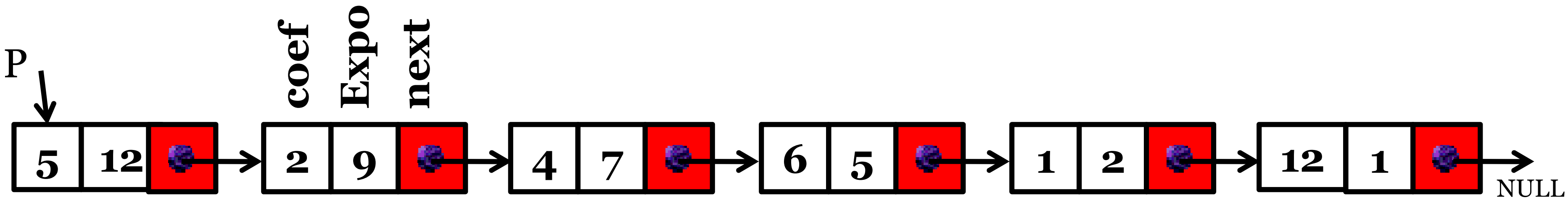
DLLs compared with SLLs

- Advantages:
  - Can be traversed in either direction (may be essential for some programs)
  - Some operations, such as deletion and inserting before a node, become easier
- Disadvantages:
  - Requires more space
  - List manipulations are slower (because more links must be changed)
  - Greater chance of having bugs (because more links must be manipulated)

Linked list Example

- **Polynomial representation:**

$$5\ x^{12} + 2\ x^{9} + 4x^{7} + 6x^{5} + x^{2} + 12\ x$$

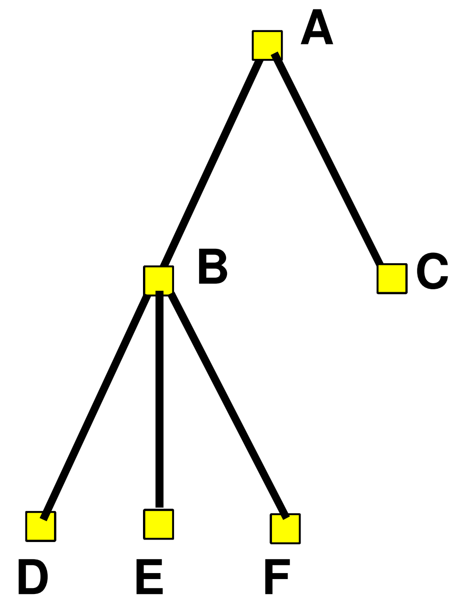


Trees

- All data structures examined so far are linear data structures.
  - Each element in a linear data structure has a clear predecessor and a clear successor.
  - Predecessors and successors may be defined by arrival time or by relative size.
- Trees are used to represent hierarchies of data.
  - Any element in a tree may have more than one successor - called it's *children*.

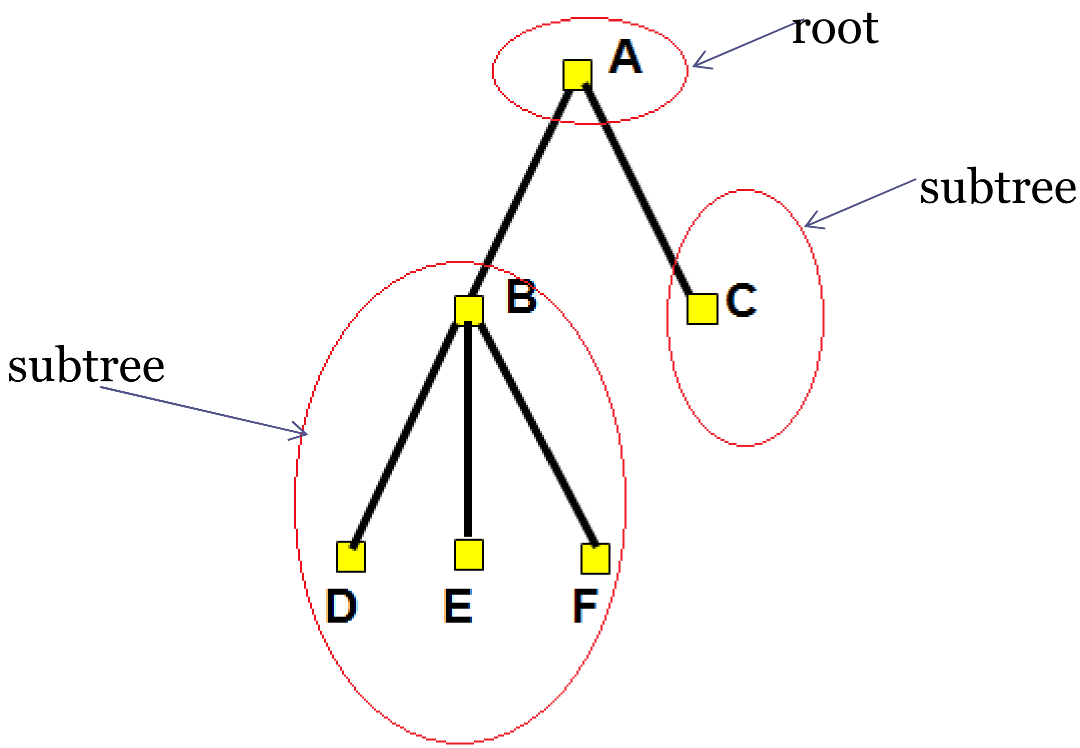
Terminology

- *Node* or *vertex* - the labeled squares
- *Edge* -the connecting lines
- In the General tree to the right:
  - B is the *child* of A - A is *parent* of B
  - B and C are *siblings*
  - A is the *root* of the tree
  - B and its children (D, E, F) are a *subtree* of A
- The parent-child relationship is generalized as *ancestor* -*descendant*.
  - B is a descendant of A - A is ancestor to B

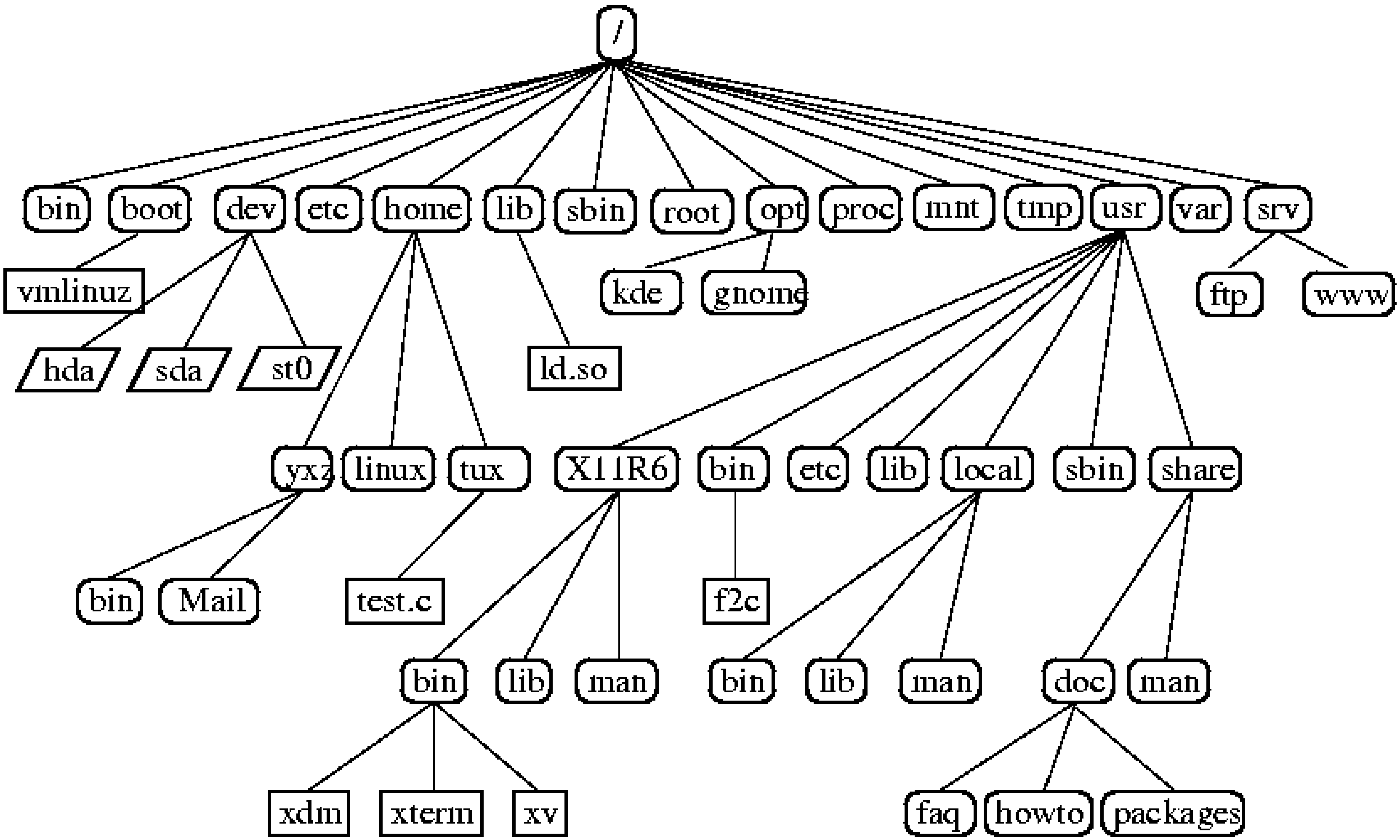


Tree: definition

- A General Tree T is a set of one or more nodes such that T is partitioned into disjoint subsets:
  - A single node R - the root
  - Sets that are general trees -the subtrees of R.

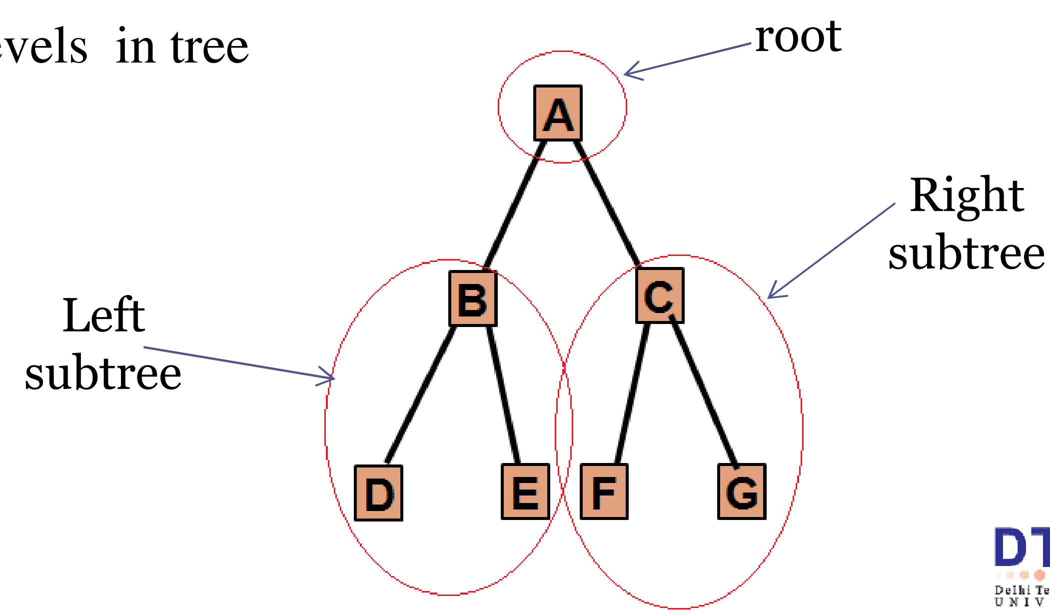


Tree example: Linux directory structure



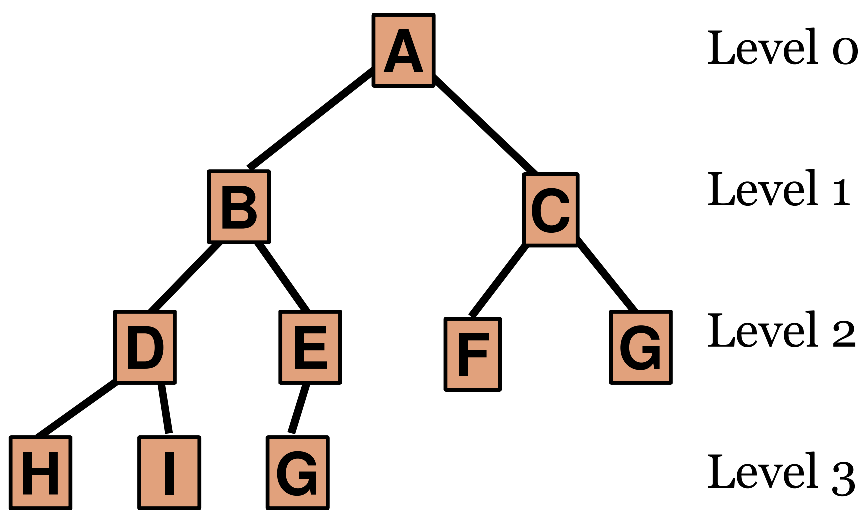
Binary Tree Definition

- A Binary Tree is a set of nodes T such that either:
  - T is empty, or
  - T is partitioned into three disjoint subsets:
    - A single node R -the root
    - Two possibly empty sets that are binary trees, called the left and right subtrees of R.
- Leaf nodes do not have any children.
- Height of tree: number of levels in tree



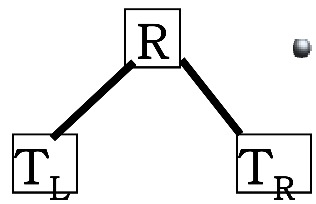
Binary tree

- Max no. of nodes in a binary tree of height h
  $= (2^h-1) = N$
- Max number of nodes at level i = 2^i
- Total number of links = 2xN
- Total non NULL links = N-1
- Total NULL links = N+1



Binary Tree Definition

- T is a binary tree if either:
  - T has no nodes, or
  - T is of the form:

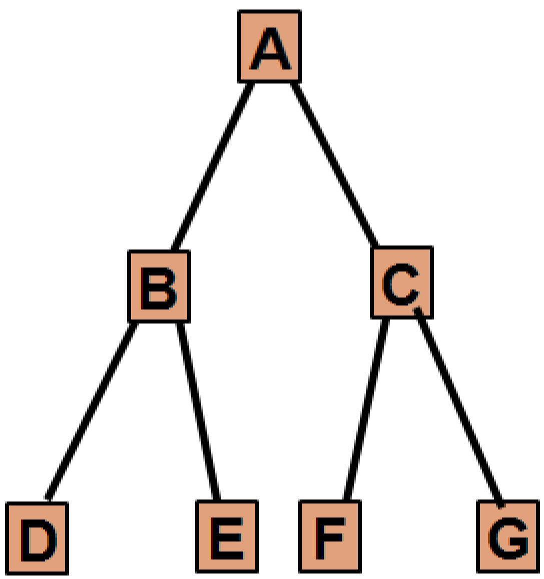


- where R is a node and T_L and T_R are both binary trees.

- if R is the root of T then :
  - T_L is the left subtree of R - if it is not null then its root is the left child of R
  - T_R is the right subtree of R - if it is not null then its root is the right child of R

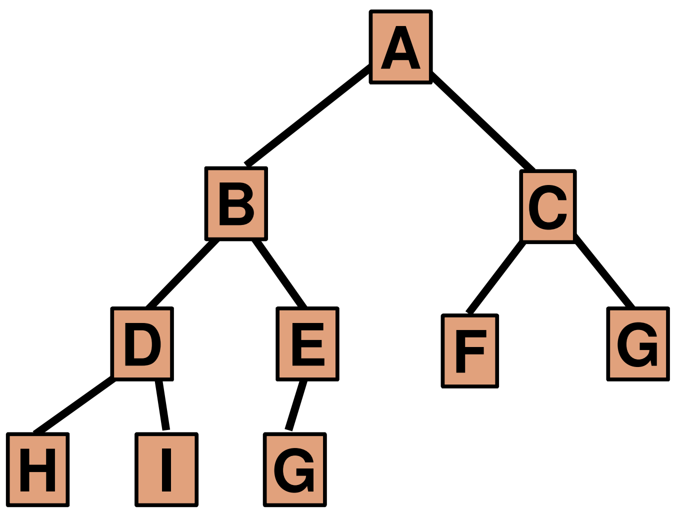
Full Binary Trees

- In a full binary tree:
  - All nodes have two children except leaf nodes.
  - All leaf nodes are located in the lowest level of the tree.
  - Height of tree = lg(N)

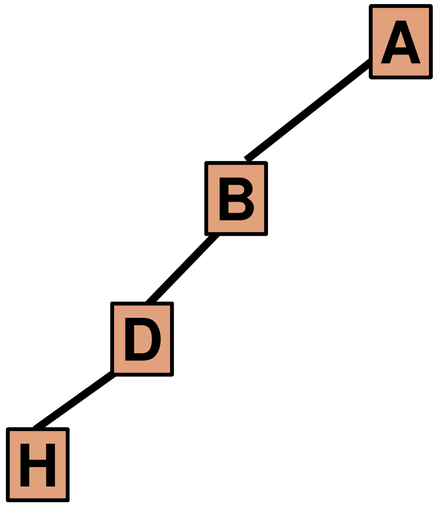


Complete Binary Trees

- In a complete binary tree:
  - All nodes have two children except those in the bottom two levels.
  - The bottom level is filled from left to right.

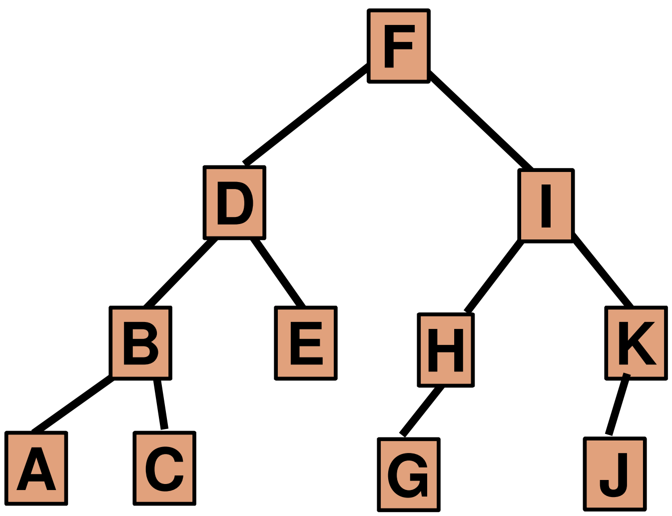


Skewed Binary Tree



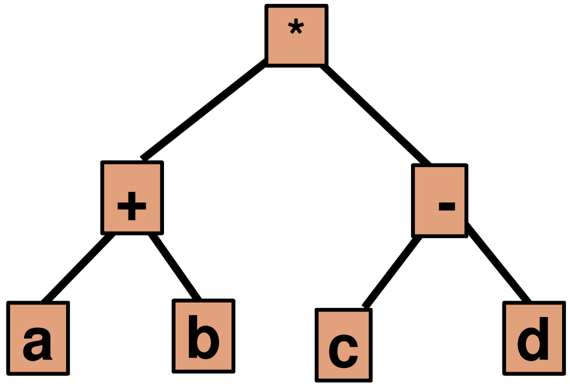
Binary Search Trees

- A binary search tree represents a hierarchy of elements that are arranged by size:
  - For any node n:
    - n's value is greater than any value in its left subtree
    - n's value is less than any value in its right subtree



Binary Expression Trees

- A binary expression tree represents an arithmetic expression:
  - For any node n:
    - if n is a leaf node:
      - it must contain an operand.
    - if n is not a leaf node:
      - it must contain an operator.
      - it must have two subtrees.



This tree represents the expression:

(a + b) * (c -d) or

a b + c d -*

Binary Tree Traversals

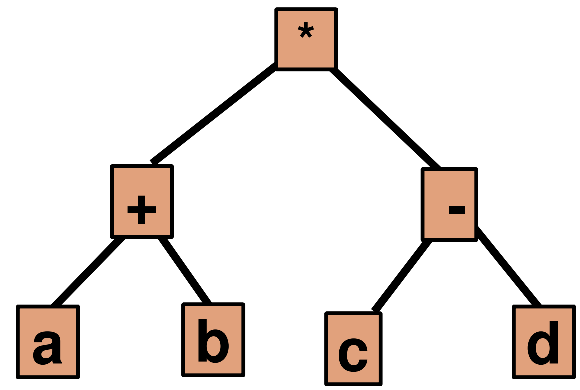
- A traversal of a binary tree "visits" each node and for each node:
  - Prints (or operates on) the data contained in that node.
  - Visits the left subtree of the node.
  - Visits the right subtree of the node.
  - Recursive termination occurs when a visited node (or tree) is null.
- There are three possible traversals of binary trees that differ only in the order that they perform the 3 basic operations.

Inorder Traversal

```
inorder (binTree tree)
{
        //performs an inorder traversal of tree
   if(tree != null)
        {
                inorder(left subtree of tree);
                print tree's data;
                inorder(right subtree of tree);
   }
}
```

Inorder Traversal

```
inorder(binTree tree){
  // inorder traversal of tree
  if(tree != null){
    inorder(left subtree of tree)
    print tree's data
    inorder(right subtree of tree)
  }
}
```



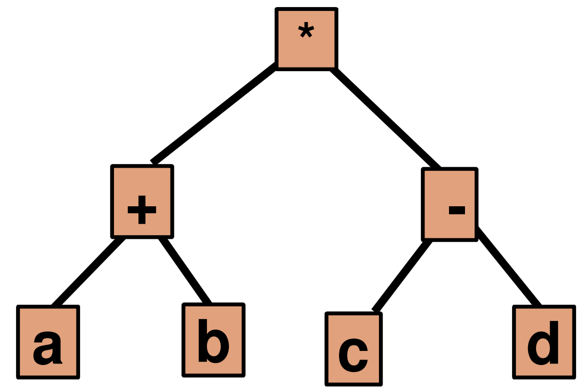
For this tree produces:
a + b * c - d

Postorder Traversal

```
postorder(binTree tree){
 //performs an postorder traversal of tree
 if(tree != null){
   postorder(left subtree of tree)
   postorder(right subtree of tree)
   print tree's data
  }
}
```

Postorder Traversal

```
postorder(binTree tree){
  //postorder traversal of tree
  if(tree != null){
    postorder(left subtree of tree)
    postorder(right subtree of tree)
    print tree's data
  }
}
```



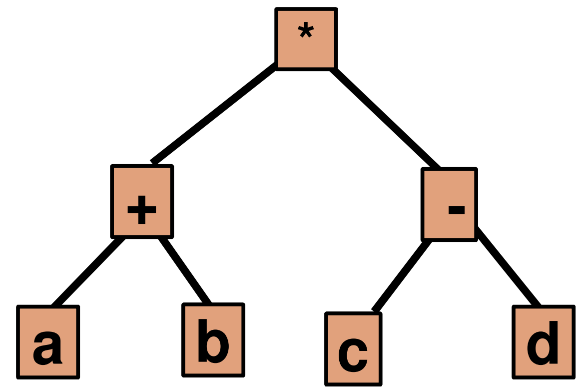
For this tree produces:
a b + c d - *

Preorder Traversal

```
preorder(binTree tree){
 //performs an preorder traversal of tree
 if(tree != null){
   print tree's data
   preorder(left subtree of tree)
   preorder(right subtree of tree)
  }
}
```

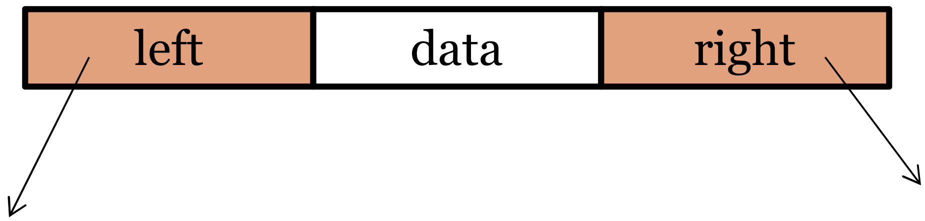
Preorder Traversal

```
preorder(binTree tree){
  //preorder traversal of tree
  if(tree != null){
    print tree's data
    preorder(left subtree of tree)
    preorder(right subtree of tree)
    }
}
```



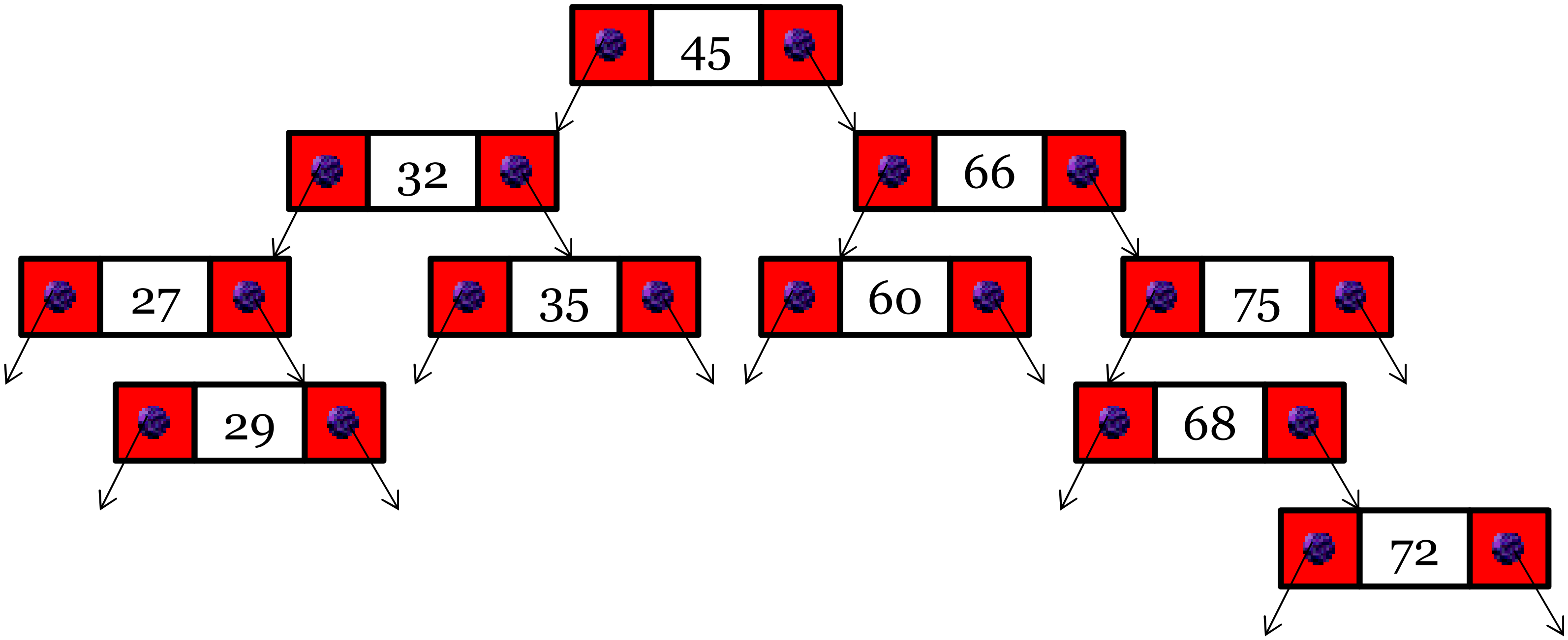
For this tree produces:
*** + a b - c d**

Binary tree: node structure

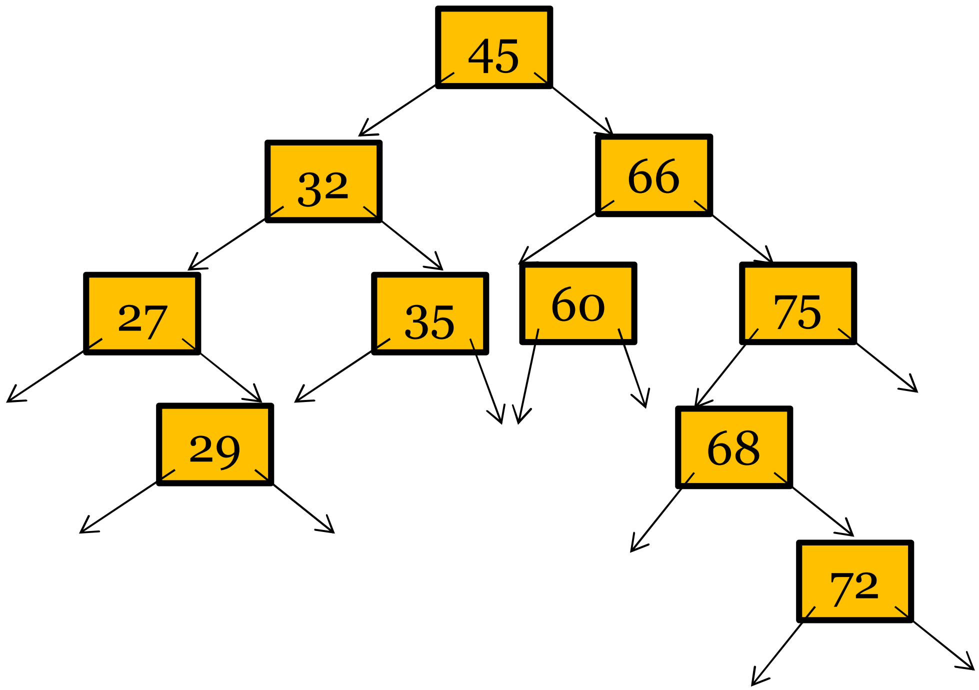



```
typedef struct bintree
{
            struct bintree *left;
            int data;
            struct bintree *right;
};
```

A binary search tree :example

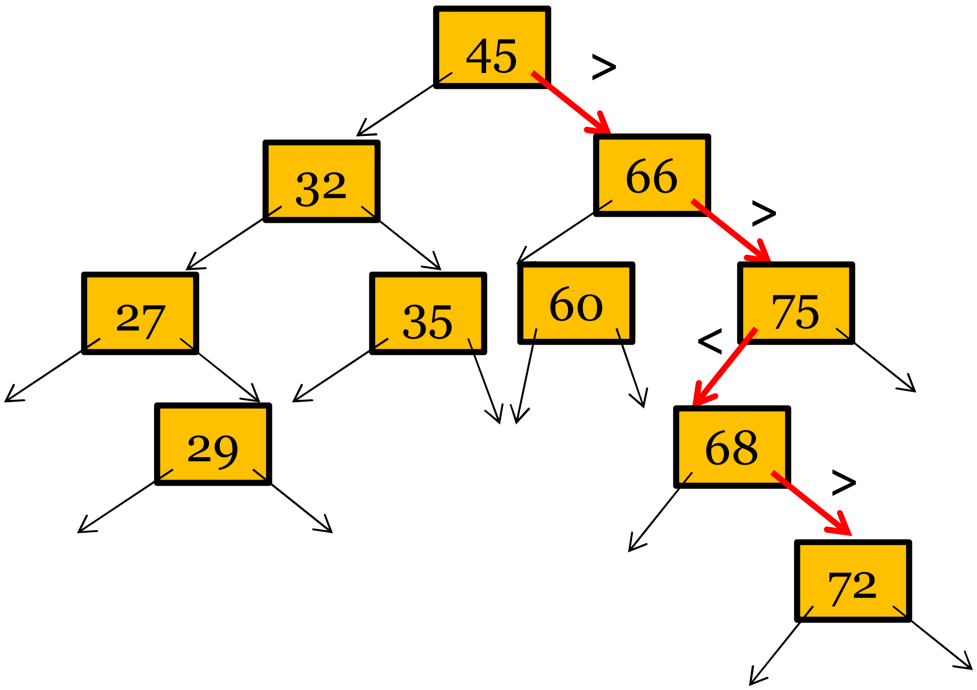


Binary search tree traversal



Inorder:	27, 29, 32, 35, 45, 60, 66, 68, 72, 75
Postorder:	29, 27, 35, 32, 60, 72, 68, 75, 66, 45
Preorder:	45, 32, 27, 29, 35, 66, 60, 75, 68, 72

Binary search tree: search

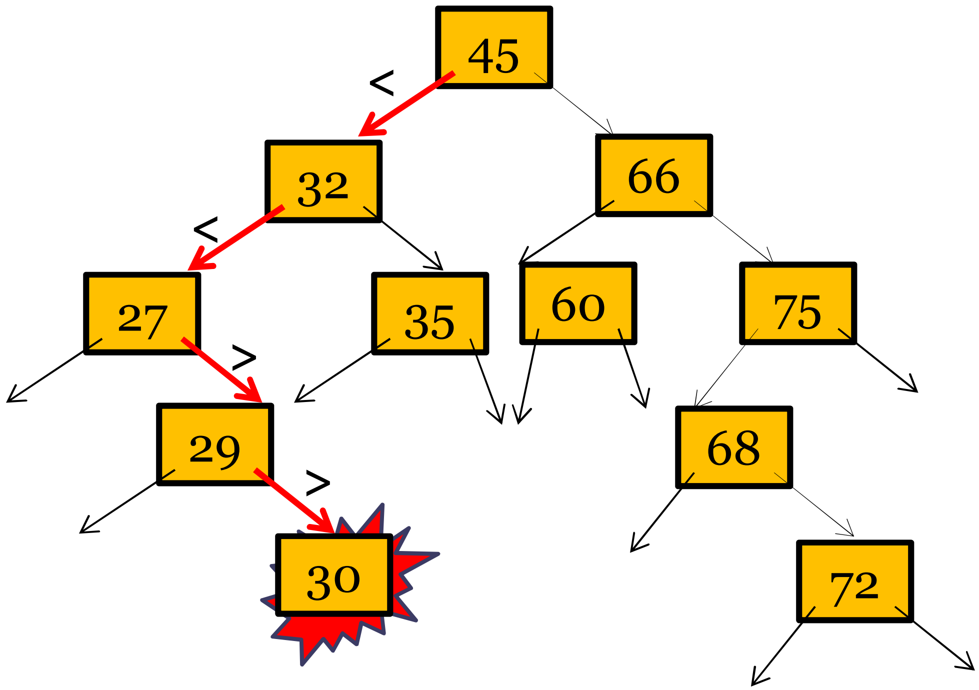


Search for 72 : 4 comparisons
Search for 74 : 4 comparisons

Search complexity: O(h)

Binary search tree: Insertion

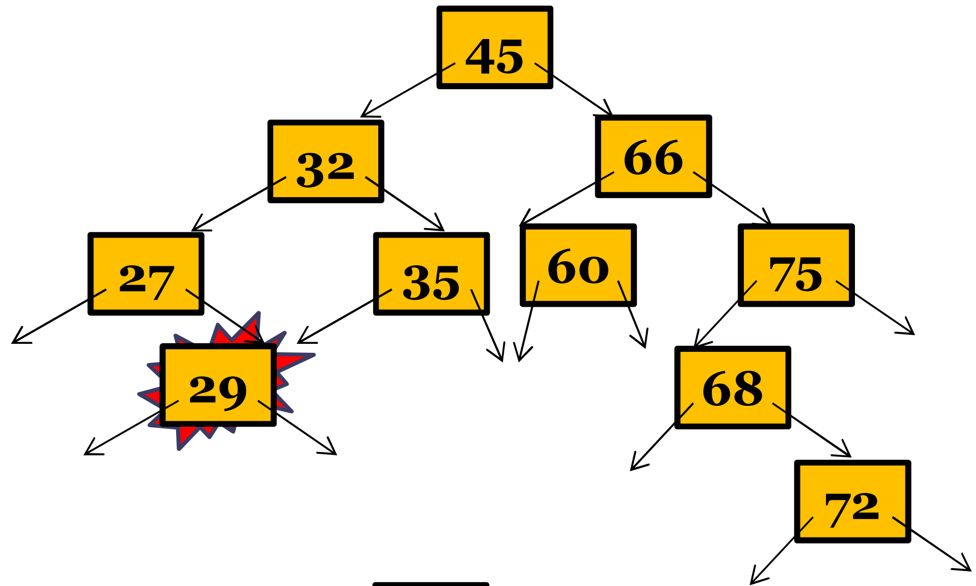
INSERT 30

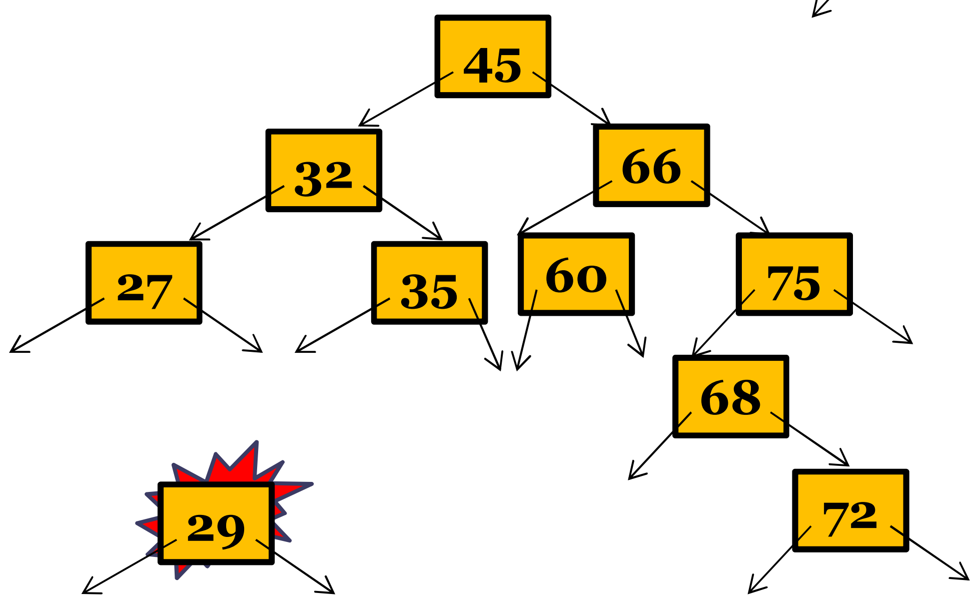


Binary search tree: deletion (Case 1)

DELTE 29
It is leaf node

Simply search it and delete

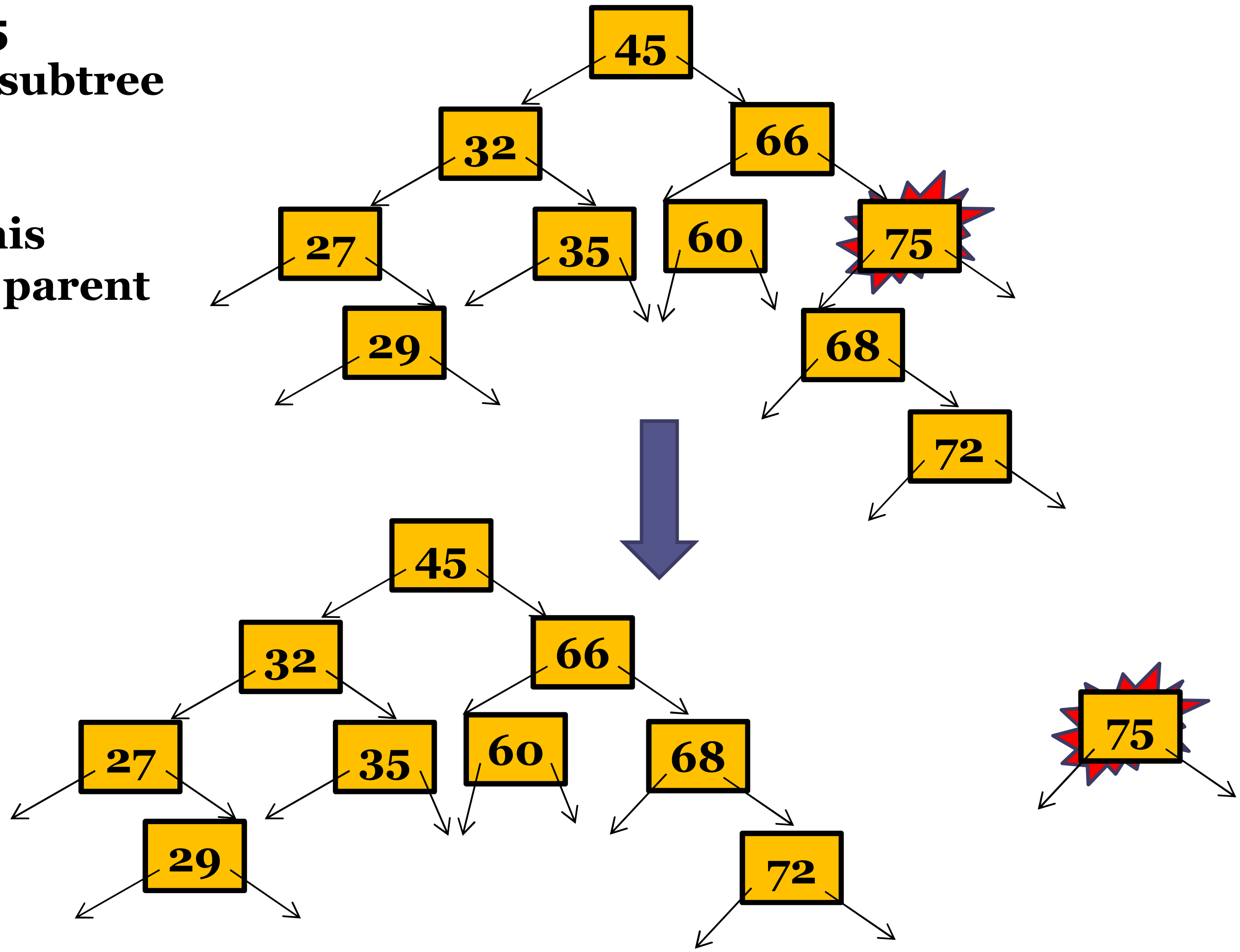




Binary search tree: deletion (Case 2)

DELTE 75
It has one subtree

Connect this subtree to parent of 75



Binary search tree: deletion (Case 3)

DELTE 66
It has both subtrees

1. **Replace data with minimum data x of right subtree of 66.**
2. **Recursively delete x from right subtree of 66.**