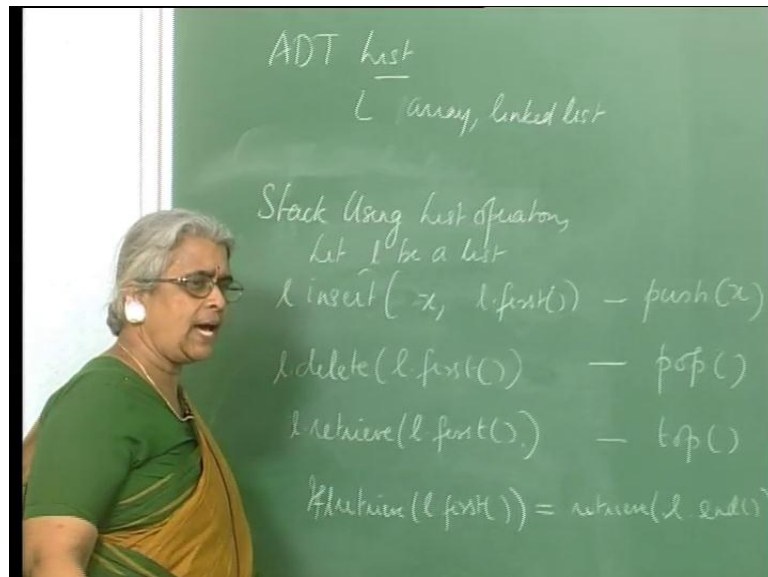


Programming, Data Structures and Algorithms
Prof. Hema Murthy
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Module – 06
Lecture - 46
Stacks: Last in first out
Operations: push, pop
Example: convert expression to postfix
Implementation of stack

Good morning class, in the last class we did the ADT call list.

(Refer Slide Time: 00:17)



And to just recap what is the ADT list? List is the sequence of items unordered items for that matter and then we defined a whole lot of operations on list. We also looked at the implementation of list, what we talk about, we talked about two different implementations although I showed you in detail link list implementation. Both the array and the link list and I showed you a link list implementation of the list. I encourage you to do the array implementation of the list. Now, I just have some problems over here, I would like you to work on storing sets I gave you an example of storing sets using list.

(Refer Slide Time: 00:58)

```
void List::printList() {
    Position p;
    p = listHead->next;
    while (p != NULL) {
        cout << p->value << " ";
        p = p->next;
    }
    cout << endl;
}
```

List Problems

- Storing Sets, performing operations on Sets (union, intersection, set A - set B)
- Infinite precision arithmetic - the entire number is represented in a list, perform operations of +, -, *, /.

And basically we want to perform operations on set union, intersection and set A minus set B.

(Refer Slide Time: 01:05)

```
p = listHead->next;
while (p != NULL) {
    cout << p->value << " ";
    p = p->next;
}
cout << endl;
}
```

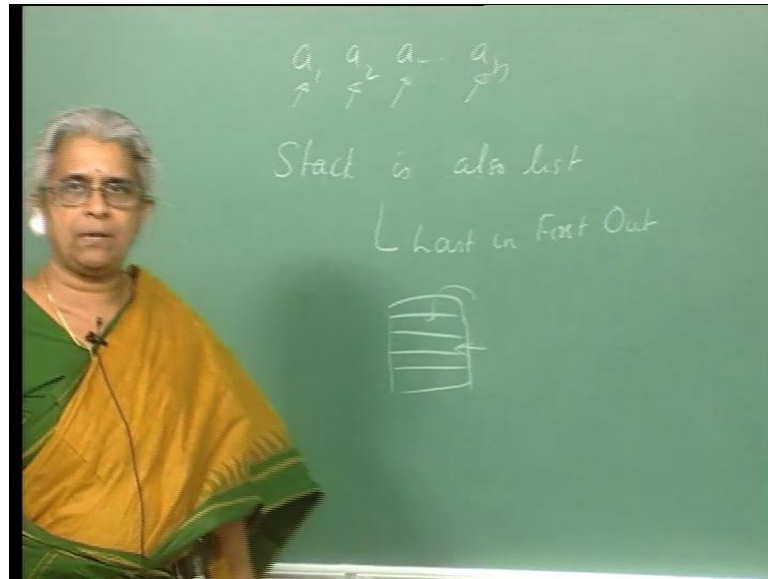
List Problems

- Storing Sets, performing operations on Sets (union, intersection, set A - set B)
- Infinite precision arithmetic - the entire number is represented in a list, perform operations of +, -, *, /.

Assignment:

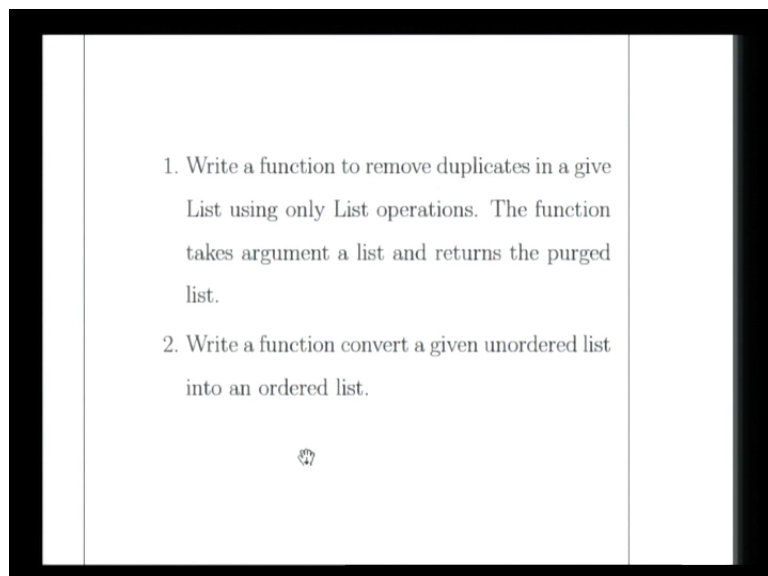
I also want you to look at another wide application of list is to or what we call infinite precision arithmetic. That is if you want to do let us say you have a 64 bit machine and you want higher precision in 64 bit, then you can use the list as a structure to perform arithmetic on it.

(Refer Slide Time: 01:32)



So, basically what we do is, what we are talking about is in the list here you are a 1, a 2, a n up to a n corresponds to different parts of the given number and then you define operations to perform arithmetic operations on this number represented as a list of element. What I mean by that is, the numbers represented in parts in terms of a 1, a 2, a n and you want to perform arithmetic operations on them. So, I would like you to try these problems.

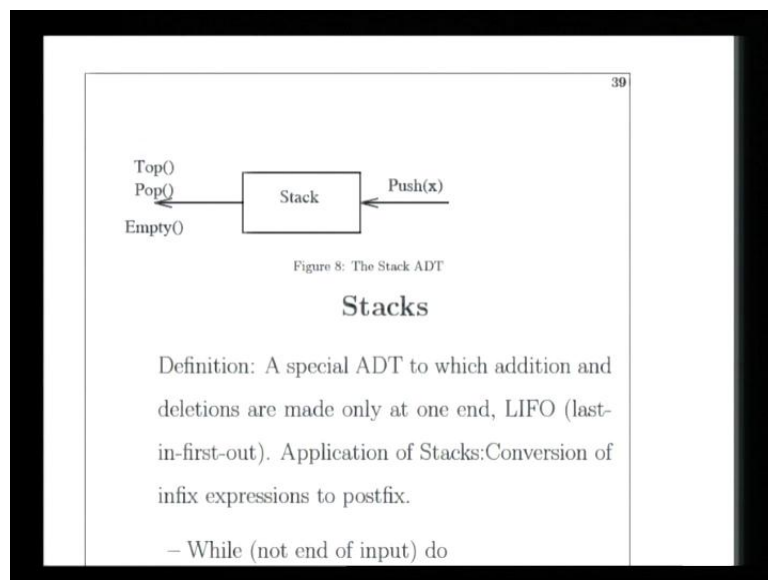
(Refer Slide Time: 02:04)



I also have a few assignment problems we talked about this a little in the last class. But, I

would like you to write a function to remove duplicates in a given list using only list operations. And basically a function which takes a list and returns the purged list and I also want you to convert a given unordered list into an ordered list, again it should be done in place. So, I would like you to do these two problems on this.

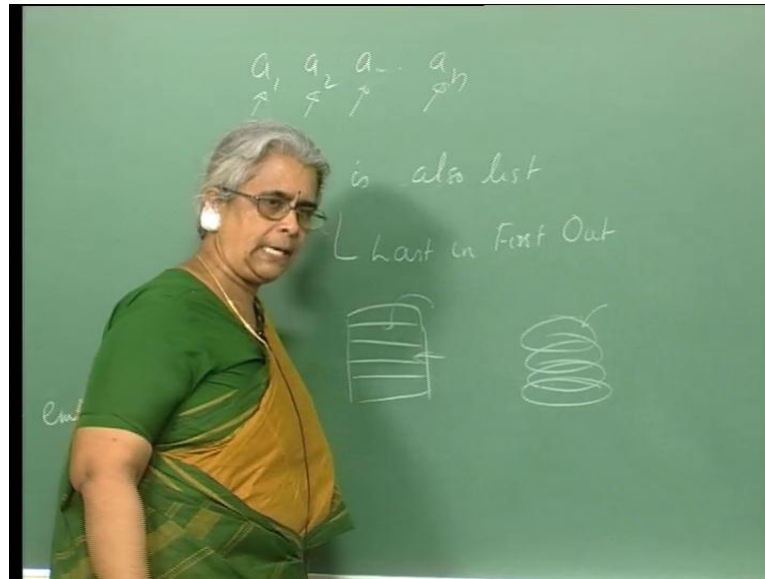
(Refer Slide Time: 02:27)



Now, next what we going to talk about is the different type of ADT called a stack. What are the stack now? Stack is also a list, but it has a fundamental difference, it is the special type of list ADT and the difference in the stack is that it operates on principles of what is called last in, first out. What is the meaning of this, that is the last element which enters into the list is removed first, this is called a stack.

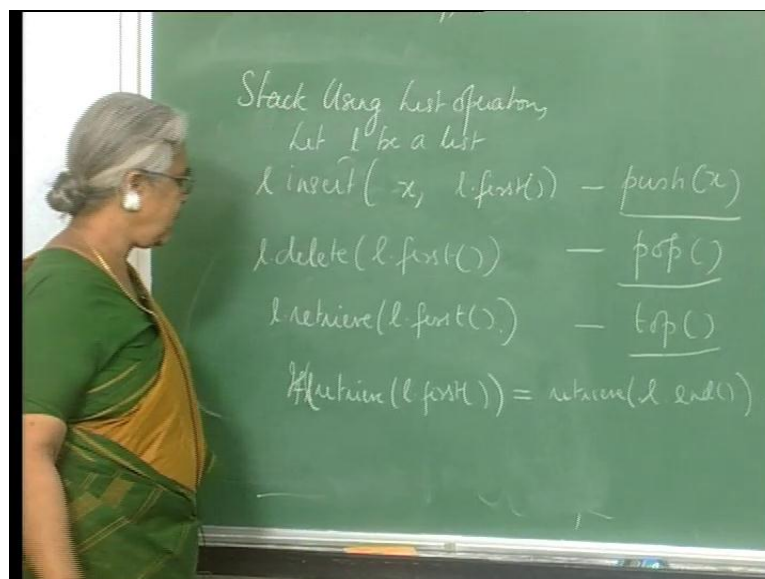
Now, let me give you some example when you putting books on your table, then you put the top most, the most recent book will be the top most book and then when you remove for example, if you want to remove this particular book, you remove the top two books before you can remove this book.

(Refer Slide Time: 03:35)



And other example let us talk about plates that are stacked in a cupboard, see put your plates over here and like this and you remove the top most plate. The most recent plate is what is first removed, if you want to remove the third plate from the top plate, you remove the first two before you can remove the third plate. So, stack has lot of nice applications and we will talk about a specific application of a stack.

(Refer Slide Time: 03:57)

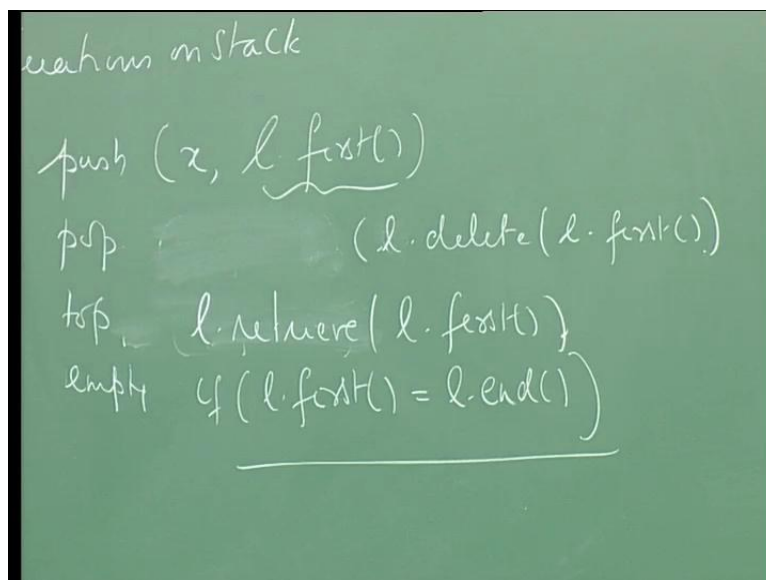


The operations that are supported by the stack are you can push, you can pop and you can find what is the top most element in the stack, can also check whether the given

stack is empty or not. So, these are the four operations that are normally supported by the stack and now since stack is a special type of list, I want to show you one example here where, since you know now the way we looked at it is I have already implemented the list ADT and since I have already implemented the list ADT, I would like to use the list itself to represent a stack.

So, how do I go about doing that the way I would do that is since the list ADT has been you know you already debugged it, you have tested it and so on and so forth. So, what we could do is we could use the list ADT itself as a stack. So, how would I do that then, then what I will do is I will restrict my operations.

(Refer Slide Time: 05:07)



So, what I will say is that the push operation, so with in my I can write an ADT stack here and says that stack or a class stack and I will say that I have l which is of type list which have I already defined it. This is my private data, then I will define a set of operations on stack and what are the operations that I want, I want push, I want pop, I want top and then I want empty. So, what will I do for push now also push x comma what will I do l dot first, what does the l dot first return, it returns the position of the first element.

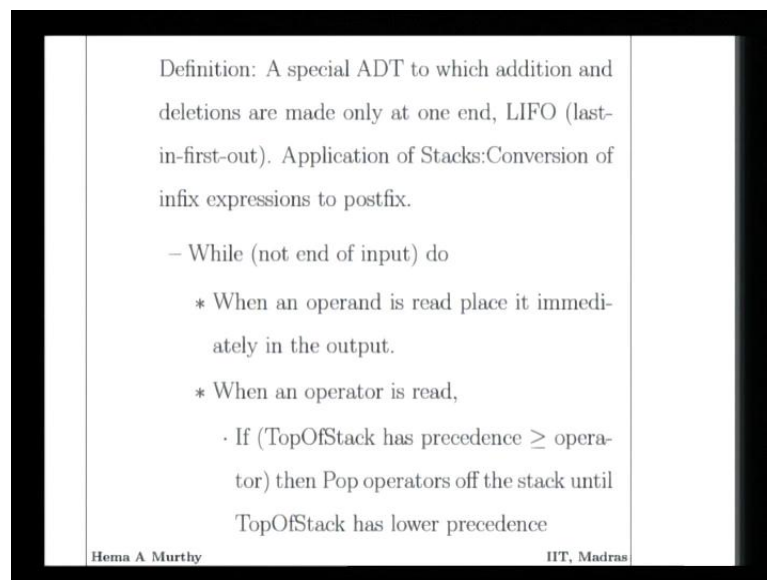
So, insert it at the first position, then I can pop, I can pop l dot first. So, this was the most recent element that was inserted, top will also give me l dot first, pop will be l dot first comma l dot delete l dot first and top is l dot first and what should be, empty is if the l dot first equals l dot end, then I can say that the stack is empty. So, what I have done this

process here now, we have all these operations over here and it should be retrieve of top should be 1 dot I would have return to be the retrieve of 1 dot first, this is what we have how you do implement all these four operations of the stack using the list.

So, you can use the list ADT in its present form which you implemented well as the stack ADT and perform the four operations which are required for the stack, now what we will do is somewhat I have this. So, this is one of the things I am trying to encourage you to do is that basically that if I also have an ADT which is implemented, I try to reuse the abstract data type as much as possible. Because, why do I want to do that the primary reason I wanted to do that is the list ADT which we implemented in the last class, we let us say nicely debugged, no errors in it.

So, what I am doing is I will quickly define a stack using the list and then I will use a stack to perform various operations. What I am going to do right now is I am going to take give you one application of stacks and then we will look at the implementation of the stack a little later.

(Refer Slide Time: 08:08)



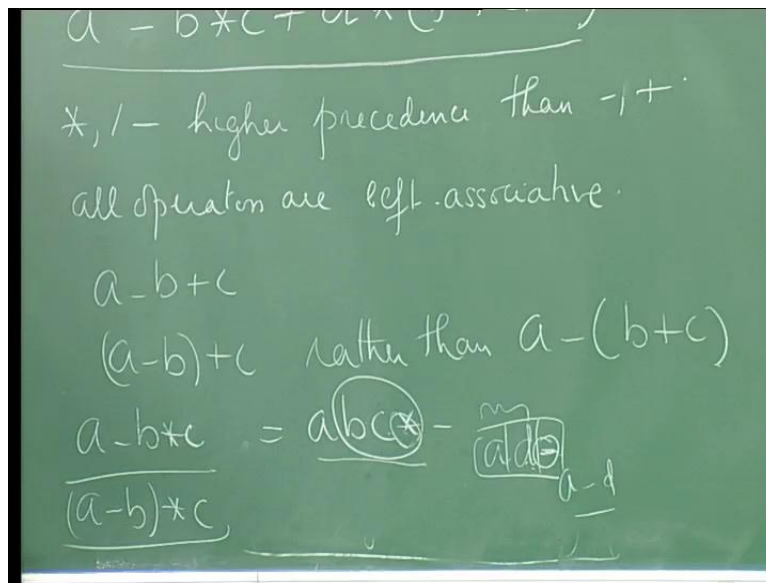
Definition: A special ADT to which addition and deletions are made only at one end, LIFO (last-in-first-out). Application of Stacks: Conversion of infix expressions to postfix.

- While (not end of input) do
 - * When an operand is read place it immediately in the output.
 - * When an operator is read,
 - If (TopOfStack has precedence \geq operator) then Pop operators off the stack until TopOfStack has lower precedence

Hema A Murthy IIT, Madras

Let us say I want to convert an infix expression to postfix expression.

(Refer Slide Time: 08:15)



Let us say why you want to do this, let us say I have an expression which is given like this. Let us say I am given an expression like this a minus b star c plus d star f plus c by d and suppose we know that star and slash had higher precedence than minus and plus and the all operators are left associative. What do we mean by this is that a minus b plus c is equal to a minus b plus c , rather than a minus of b plus c that is we assume that all the operators associate to the left and this is a meaning of it.

I am given a expression like this I do not know whether I have to perform a minus b plus c or should I perform a minus of b plus c that is where the associativity matters. What is the interesting is that when I am given an expression like this and I do not know the associativity of the operators, it is difficult to evaluate this expression.

So, what is done is given a particular expression, the expression is converted to a form called the postfix expression. So, let us see what this postfix expressions is all about, let me just take only a partial expression a minus b star c let us say, this will be written as a, b, c star minus. What is the meaning of it, we take the top most operator. Then, you come to the next operator and the next one is still an operator, so you push it somewhere, take the next operator, again you use the stack for this. Then, you find on the top there are two operators over here, two operands over here you compute this b star c .

The idea is that what happens is the operator associate with the some binary operator associates with the nearest operands that are there. So, what is nice is the advantage of

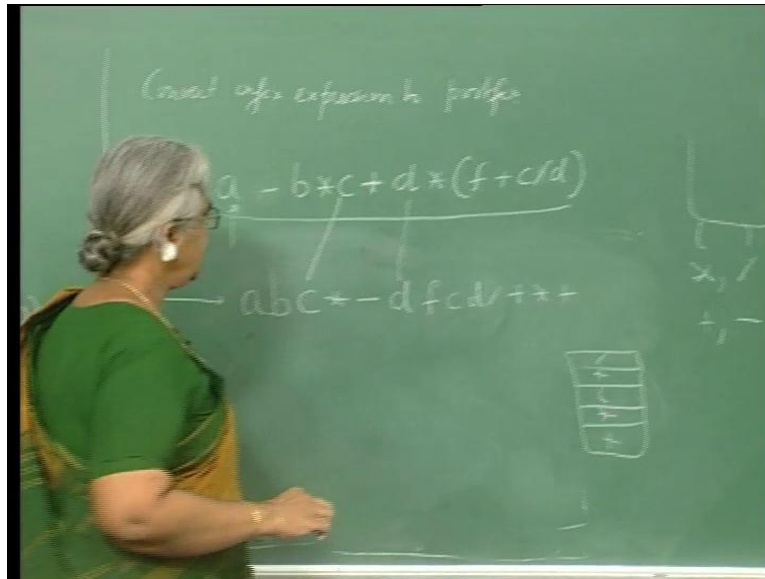
this expression when you write it in this form, there is no issue about the evaluation. Why is there no issue about the evaluation? Because, you look at the operator, you just for example, if I keep moving along this as soon as I come across the first operator, all that I will do is I take this operator, operated on these two operands get the result and put it there and that we call it d, then I have a d and minus. Then, it means after find I am assume a minus d.

So, that is absolutely no ambiguity when a given expression is given in postfix, the order of the operations. Whereas, if I give the expression in this form which is called infix, I do not know whether I am has to perform a minus b star c, I do not know the associativity, I do not know the precedence of the operators. Everything is subsumed in the postfix expression. Given a postfix expression all I have to do is I traverse that postfix expression from left to right, as soon as I come across an operator I take the nearest two operands, perform the computation, then you can put it back on stack, this is the way of evaluating again using the stack.

Then, once again the result I have a d and minus on the particular stack, then again when I traverse this for example, next I see another operand over here and as soon as I see another operator over here, then I take the most nearest two operands, perform this operation and I get the result. So, the evaluation of expressions is normally done using the postfix expression. You take an infix, infix expression is very convenient for us to write.

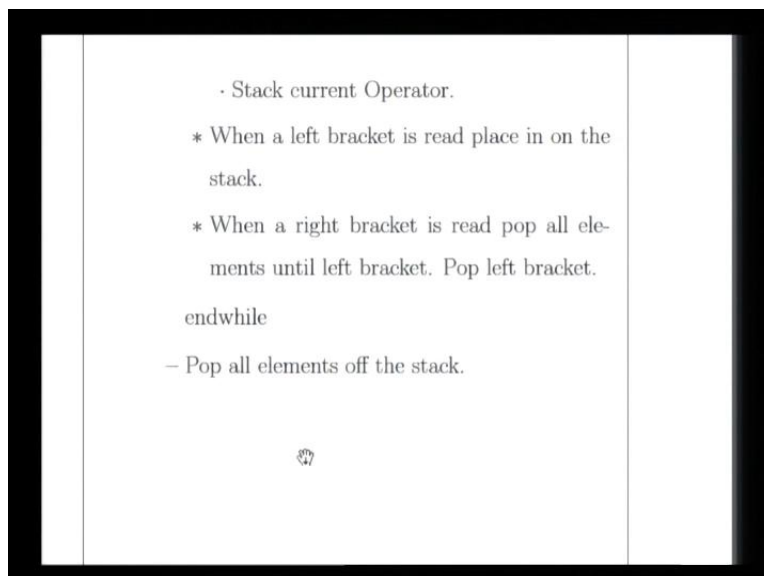
But, what we will do is we take the infix expression, convert the infix expression to a postfix expression and then perform the computation. So, now what I am going to show you is the evaluation also can be done using stacks, but it will take one application. We look at conversion of infix to postfix using the stack, let us see how you get this done. So, what is it done here, when an operand is read that is I am getting this expression here as soon as I find an operand, what am I going to do I am just going to simply place it on the output.

(Refer Slide Time: 14:18)



Then, when an operator is read, what do I do I take this operator, so that means, let us go back to this particular expression, take this expression now I see an operand I put it, this is my output, this is my input, then what do I do I put it to the output. When an operators read, what I do is I look at the element I have a stack here, right now my stack is empty. Look at the element which is there one on the top of the stack, if the top of the stack has higher precedence, then I keep popping all the elements from the top of the stack.

(Refer Slide Time: 14:58)



So, this part is not that, because it is empty, so what I am going to do, I am going to

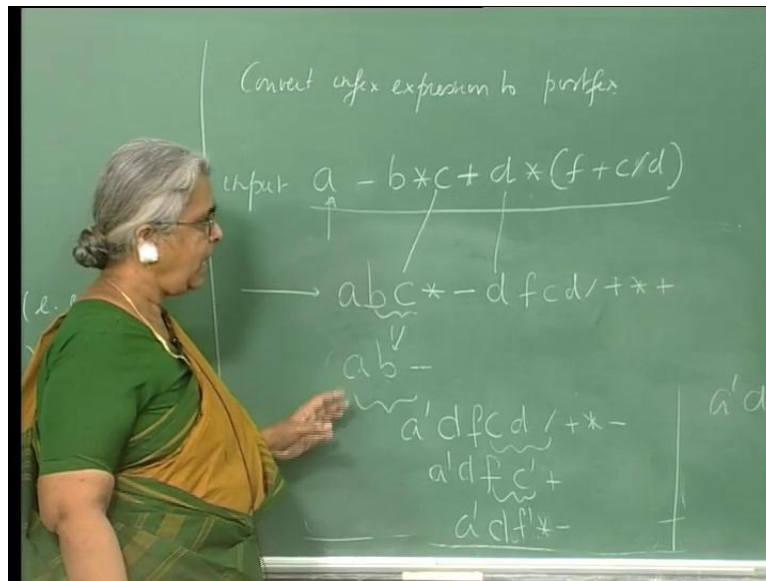
simply stack the current operator, I put minus 1 on it, then I have b here then what do I see, then I have a another I see another operator over here. So, now I look at the top of the stack this is a minus sign here, minus sign has lower precedence. So, what I do I just simply stack star also on to it, then again I see an operand move it to the output. Next what do I see, I see a plus, when I see a plus what I do is the following, I look at the content on the top of the stack.

So, star is the content on the top of the stack, so since star is a content on the top of the stack I output and star has higher precedence. So, the precedence is like this star comma slash and plus comma minus and of course, above this you have the brackets. So, what I do is I pop the star. Then, once since I pop the star what do I have, I only have minus on top of the stack. What the current operator that I am looking at plus, now clearly plus and minus have same precedence.

So, what I do I also pop the minus, then what I do I put this plus over here onto the stack and with precedence, this is how this whole operation course about. So, basically this is what we do and now I have done, up till this part of the expression. Now, let us look at the next one what do I have now, the plus is gone it is gone to the stack, I get d I push this on to the stack. Then, what do I have, I have a left bracket I put the left bracket also on to the stack.

Then, what do I have, I have a right, then I put the plus on to the stack, then I have c d, then I have a slash over here and compare with what is there, put it on the top of the stack, then what happens I come across the right bracket. So, as soon as we come across the right bracket I pop everything of until I see the left bracket that mean I put the slash here, I put the plus here and then what do I do, I come to the end of the expression star and plus. So, this is how I complete the expression.

(Refer Slide Time: 17:56)



So, this is the conversion from infix to postfix. So, what we have done, we took the infix expression, then we converted it to a postfix expression using a stack. Now, let us see if it will give us the evaluation correctly, what did I say. I just keep this as it is, then as soon as I see an operator I find the nearest two operands and compute the computation. Let me call this b operand that is I computed these two and let us say this result is b prime, then what do I have, I have a b minus.

Now, clearly there are two operators and there are two operands and there is an operator, so I compute this let me call this a prime. So, what do I have now, I have already computed this, now I have d f c d slash plus star plus, so what do I do now, I keep moving from left, this is computed let me call it is c prime, then I have f c prime d a prime. So, what do I do, I compute this let me call this f prime, so I have a prime d, f prime plus and plus is already gone, d prime plus f plus c d.

So, let me call this c prime plus is already gone, I already computed this sum over here and then I have a star over here and a minus. So, next what do I do, I compute a prime d f prime let me call it d prime after the multiplication, because I see the multiplication here, multiply these two put the result in d prime I have a minus and now finally I and get the result which is essentially what the...

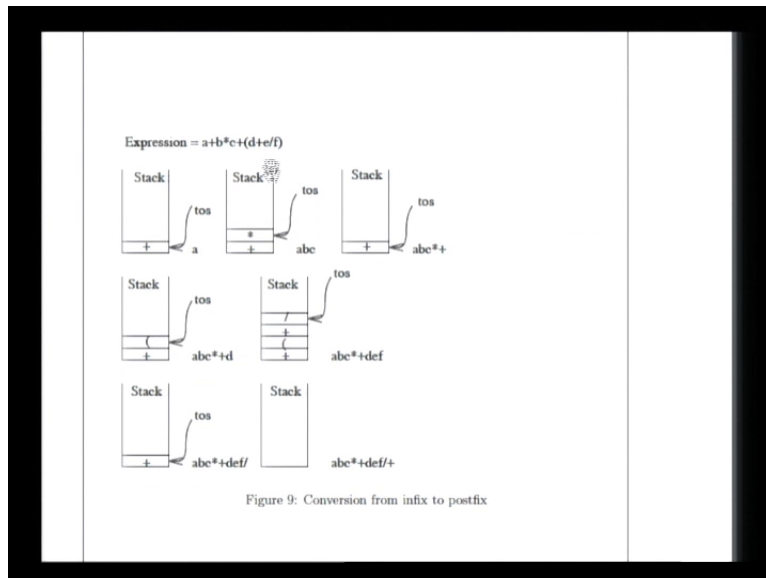
So, basically what it tells is if I follow the rules of arithmetic for a postfix gives expression, what is a rule say I take the... whenever I see an operator, I take the nearest

two operands, perform the computation and store it. Then, what happens that becomes a new operand now b' , then a and b' and this is an operator here minus. So, I compute the difference of a and b' , again I put it back on top. So, then I call it a' , then what happens I have $d f c d$, I just put them all together, then I see a slash.

When I see a slash what do I do, I take the nearest two operands, multi divide them, here the operators slash, so I divide c by d I call it c' . Then, I have $a' d f c'$, then what happens I have f and c' . So, this is this part, so I add them then what do I had $a' d f'$ let me call it, then $d f'$ star this star is here. So, I perform this computation and because that is the first operator that I am seeing I take the nearest two operands perform the computation, then finally I get let me call it I got a new operator, operand called d' and $a' d'$ and I subtracts one from the other and the postfix expression can be evaluated.

So, clearly what it tells me is using the algorithm that I have given here, the conversion to infix to postfix is correct. So, basically I can use a stack to take this input convert it to postfix. So, whatever you doing when you doing this, we are taking all the operators, stacking the operators including brackets and when do we pop as soon as we come across an operator which has lower precedence than what is there in the stack, I pop all the operators from the stack which have higher precedence and the operators which have the same precedence too and put this operator on to the stack that is exactly what we did. Let me illustrate this with the pictorial example again.

(Refer Slide Time: 21:57)



So, let us say we have the same I have a same almost the same expression which I have already gone through over here and never realize it. This is the same expression that is there and here is a pictorial description of what is exactly happening stack the plus, this is the top of the stack, then a is kept outside and then you get a, b, c then you put a, b, c star, pop this and so on. So, this is how your output gets generated. As soon as you see a bracket you just push everything out. So, this is how you convert and invert...

So, basically this is the first thing whether the stack is a stack of operators. So, what is happening now plus and star has stack, then as soon as the next plus comes you pop both plus and star, just I was already told to you and then you see the bracket, you put the bracket on top of the stack and repeat the same process again. So, this is essentially how stack can be used.

(Refer Slide Time: 22:54)

```
Implementation of Stacks

/* begin (Definition of class stack) */
#define STACK_SIZE 256
class Stack {
    char string[STACK_SIZE];    // A Stack of characters
    int tos;                    // A pointer to the top of Stack.
public:
    void makeNull();           // creates an Empty Stack
    void push(char x);         // pushes an element on to the Stack
    char top();                // returns the element on Top of the Stack
    char pop();                // Pops an element from the Stack
    int empty();               // returns 1 if Stack is empty
};
/* end(Definition of class Stack) */

/* begin(Implementation of the class Stack) */
void Stack::makeNull() {
    tos = STACK_SIZE;
}
}
```

Now, let us look at the implementation of the stack and what are the operations, I am looking at a separate implementation. Let us say that I am not going to use the list, I want to implement my own stack. Before that what are time complexity of these operations when look at I dot insert, I dot delete, I dot retrieve, what will be the time complexity of all of these, if you can see if you look at the, it depends upon the kind of implementation.

If the implementation is a linked list it does not matter, doing this particular case I am inserting at the first, all these operations can also be done in order one type, even when I am using the list. But, we can do a cleaner implementation, we can implement a stack. So, since that in this particular example I am using the operators I have created a stack which is I am using an array implementation here, this is your static array.

So, basically the stacks size here, see you would also like to have a stack full perhaps. So, I create or you resize the stack, if you want it to be transparent you can always resizes the stack. Then, you create an empty stack which is make null, then you push the character on to the stack, character pop returns the top most element on the stack, pop deletes the element from the top of the stack, empty tells you that the stack is empty. From here is how the various operations can be performed and basically what is interesting is all the operands, operations here, if you look at this.

(Refer Slide Time: 24:37)

```
#define STACK_SIZE 256
class Stack {
    char string[STACK_SIZE];    // A Stack of characters
    int tos;                    // A pointer to the top of Stack.
public:
    void makeNull();           // creates an Empty Stack
    void push(char x);         // pushes an element on to the Stack
    char top();                // returns the element on Top of the Stack
    char pop();                // Pops an element from the Stack
    int empty();               // returns 1 if Stack is empty
};
/* end(Definition of class Stack) */

/* begin(Implementation of the class Stack) */
void Stack::makeNull() {
    tos = STACK_SIZE;
}
void Stack::push(char x) {
    tos--;
    string[tos] = x;
}
}
```

If you look at the analysis of this I want you to go through this over here, make null it is simply creates the top of the stack to be stack size. Then, we just putting from pushing from the stack, what are we doing we here in push for example, I making the stack size, the top of stack initially match to the largest element possible in the array, largest index possible in the array, then what we do, we decrement the top of the stack and then push the element there and then when we... So, this is again an order one operation if you notice, this is an order one operation, this is an order one operation.

(Refer Slide Time: 25:17)

```
char Stack::top() {
    if (tos < STACK_SIZE)
        return string[tos];
    else
        return(0);
}
char Stack::pop() {
    char tmp;
    if (tos >= STACK_SIZE) return 0; else {
        tmp = string[tos];
        tos++;
        return tmp;
    }
}
int Stack::empty() {
    if (tos >= STACK_SIZE)
        return 1;
    else
        return 0;
}
/*end (Implementation of class Stack) */
```


Then, what we are doing if the top of stack is less than stack size, it is returning the top of stack that mean, otherwise what is happening when you performing a top operation, that is no element in the stack that is the meaning of this return 0 over here. Then, if top of stack greater than stack size returns 0 again; otherwise, what are we doing you are popping the particular element on the stack, that is when you are trying to pop more elements than there are, otherwise if there is an elements on the stack, it pops this.

(Refer Slide Time: 25:47)

```
char    tmp;
if (tos >= STACK_SIZE) return 0; else {
tmp = string[tos];
tos++;
return tmp;
}
}
int Stack::empty() {
if (tos >= STACK_SIZE)
return 1;
else
return 0;
}
/*end (Implementation of class Stack) */
```

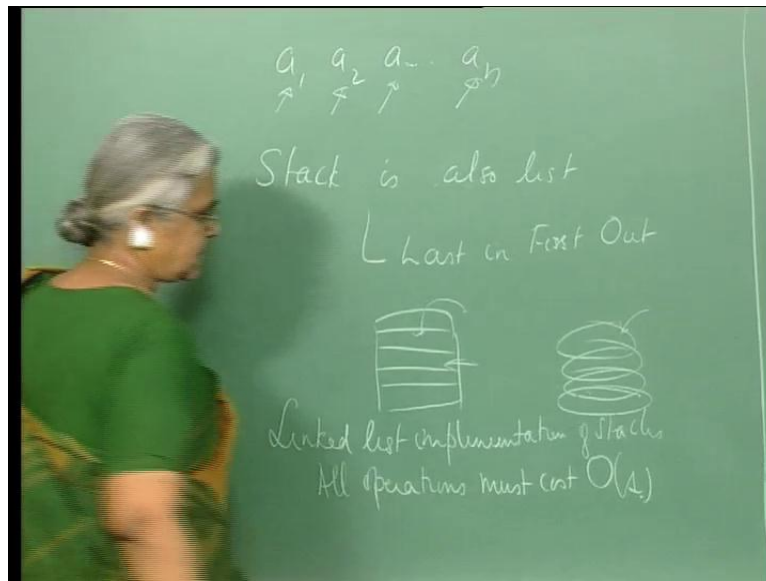
Time complexity of Stack operations using arrays
- $O(1)$

Assignment:

Hema A Murthy IIT, Madras

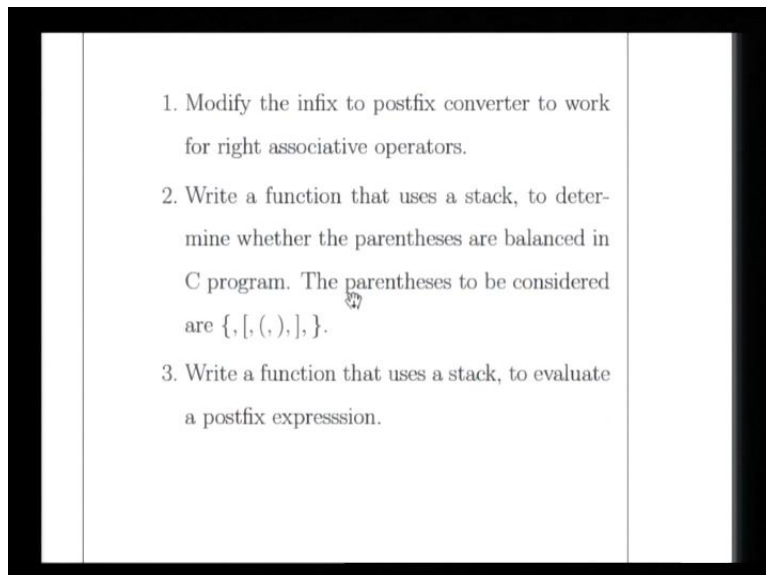
Then, if stack is empty, it is top of stack is greater than stack size, then you say return greater than or equal to stack size, then you say that the stack is empty. For the time complexity of all the operations, there are performance stacks using array is order one.

(Refer Slide Time: 26:12)



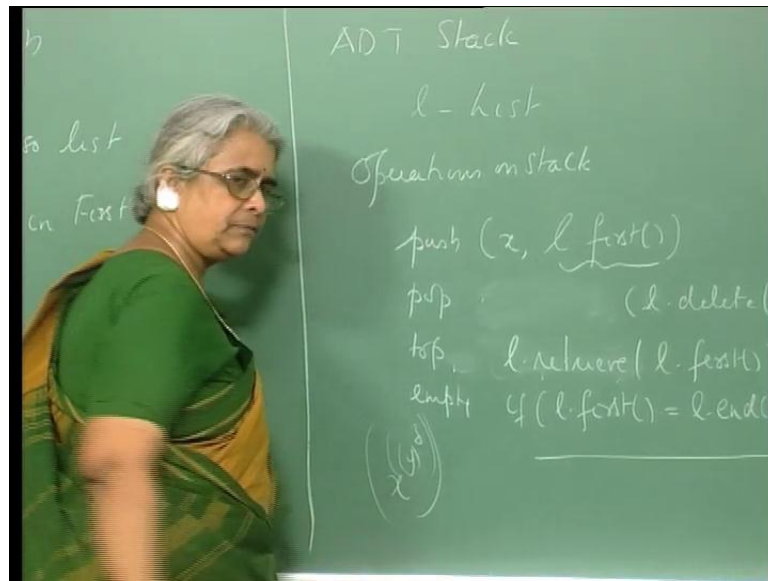
Now, what I want you to do is I want you to go back and implement the stack, I given you an array implementation of stacks I want you to do a link list implementation of stacks and all operations must cost the order one. So, I want you to do go back and experiment this. Now, what I want to do is I will just close this with some applications.

(Refer Slide Time: 26:46)



Now, I want you to do this, right now we looked at operations, we only operators we only looked at left associative operators. What happens if an operator associates to the right, let me give you an example.

(Refer Slide Time: 27:04)



If I had an example like this x to the power of y to the power of z , then what is this, this is essentially y to the power of z whole x to power of. So, I want you to think of this how would you use is stack, how would you modify the infix to postfix converter, if you worked with operators which are right associative. The other function that I would like you to write is to determine whether the parentheses are balanced in a C program. I want you to look at all these types of parenthesis. I also want you to write a function that uses a stack you already talked about it to evaluate a postfix expression.