Made by vu £lon musk files

CS301 Mid term Notes

Lec 1 to 22

Data Structures Lecture No. 01

Data structures

Data structures are used to store data in a computer in an organized form. In C++ language, Different types of data structures are; Array, Stack Queue, Linked List, and Tree. Sorting technique is merge sort, shell sort, bubble sort, Quick sort, Selection sort, Heap sort etc.

Organizing Data

- Any organization for a collection of records that can be searched, processed in any order, or modified.
- The choice of data structure and algorithm can make the difference between a program running in a few seconds or many days.

Data Structures

Data Structures are a specialized means of organizing and storing data in computers in such a way that we can perform operations on the stored data more efficiently.

Most Common 8 Data structures are as follows:

1. Arrays
2. Linked Lists
3. Stacks
4. Queues
5. Hash Tables
6. Trees
7. Heaps
8. Graphs

vu £lon musk files

Arrays

An ordered and index able sequence of values. C++ supports arrays of a single dimension (a vector) or of multiple dimensions. Elementary data structure that exists as built-in in most programming languages.

```
#include <iostream>

using namespace std;

main( int argc, char** argv )

{

        int x[6];

        int j;

        for(j=0; j < 6; j++)

        x[j] = 2*j;

        }
```

Array operations

1. Traverse: Go through the elements and print them.
2. Search: Search for an element in the array. You can search the element by its value or its index.
3. Update: Update the value of an existing element at a given index.

Inserting elements to an array and deleting elements from an array cannot be done straight away as arrays are fixed in size. If you want to insert an element to an array, first you will have to create a new array with increased size (current size + 1), copy the existing elements and add the new element. The same goes for the deletion with a new array of reduced size.

Applications of arrays

vu elon musk files

- Used as the building blocks to build other data structures such as array lists, heaps, hash tables, vectors and matrices.
- Used for different sorting algorithms such as insertion sort, quick sort, bubble sort and merge sort.

The LIST Data Structure

- The List is among the most generic of data structures.
- Real life:
    a. shopping list,
    b. groceries list,
    c. list of people to invite to dinner
    d. List of presents to get

Lists LONG OR SHORT OR Quiz

- A list is collection of items that are all of the same type (grocery items, integers, names).
- The items, or elements of the list, are stored in some particular order.
- It is possible to insert new elements into various positions in the list and remove any element of the list.
- List is a set of elements in a linear order.
  For example, data values a_1, a_2, a_3, a_4 can be arranged in a list:

  (a_3, a_1, a_2, a_4)

  In this list, a_3, is the first element, a_1 is the second element, and so on
- The order is important here; this is not just a random collection of elements, it is an *ordered* collection.

List Operations FOR MCQ

vu Elon musk files

Useful operations

- create List(): create a new list (presumably empty)
- copy(): set one list to be a copy of another
- clear(); clear a list (remove all elements)
- insert(X, ?): Insert element X at a particular position in the list
- remove(?): Remove element at some position in the list
- get(?): Get element at a given position
- update(X, ?): replace the element at a given position with X
- find(X): determine if the element X is in the list
- length (): return the length of the list.

What is meant by "particular position" we have used "?" for List Operations?
There are two possibilities:

1. Use the actual index of element: insert after element 3, get element number 6. This approach is taken by arrays.
2. Use a "current" marker or pointer to refer to a particular position in the list. If we use

The "current" marker, the following four methods would be useful:

start(): moves to "current" pointer to the very first element.
tail(): moves to "current" pointer to the very last element.
next(): move the current position forward one element.
back(): move the current position backward one element.

Lecture No. 02 Data Structures

Linked Lists

vu Elon musk files

A linked list is a sequential structure that consists of a sequence of items in linear order which is linked to each other. Hence, you have to access data sequentially and random access is not possible. Linked lists provide a simple and flexible representation of dynamic sets. Let's consider the following terms regarding linked lists.

• Elements in a linked list are known as nodes.

• Each node contains a key and a pointer to its successor node, known as next.

• The attribute named head points to the first element of the linked list.

• The last element of the linked list is known as the tail.

Following are the various types of linked lists available.

• Singly linked list — Traversal of items can be done in the forward direction only.

• Doubly linked list — Traversal of items can be done in both forward and backward directions. Nodes consist of an additional pointer known as prev, pointing to the previous node.

• Circular linked lists — Linked lists where the prev pointer of the head points to the tail and the next pointer of the tail points to the head

Following are the various types of linked lists available.

• Singly linked list — Traversal of items can be done in the forward direction only.

• Doubly linked list — Traversal of items can be done in both forward and backward directions. Nodes consist of an additional pointer known as prev, pointing to the previous node.

• Circular linked lists — Linked lists where the prev pointer of the head points to the tail and the next pointer of the tail points to the head. Linked list operations

• Search: Find the first element with the key k in the given linked list by a simple linear search and returns a pointer to this element.

www.elonmusk.files

• Insert: Insert a key to the linked list. An insertion can be done in 3 different ways; insert at the beginning of the list, insert at the end of the list and insert in the middle of the list.

• Delete: Removes an element x from a given linked list. You cannot delete a node by a single step.

A deletion can be done in 3 different ways; delete from the beginning of the list, delete from the end of the list and delete from the middle of the list.

Applications of linked lists

• Used for symbol table management in compiler design.

• Used in switching between programs using Alt + Tab (implemented using Circular Linked List).

List Implementation using Methods

I. add Method
II. next Method
III. remove Method
IV. find Method
V. Other Methods

List Implementation

In C language, a linked list can be implemented using structure and pointers. Strict Linked List {int data; struct Linked List *next; }; The above definition is used to create every node in the list. The data field stores the element and the next is a pointer to store the address of the next node.

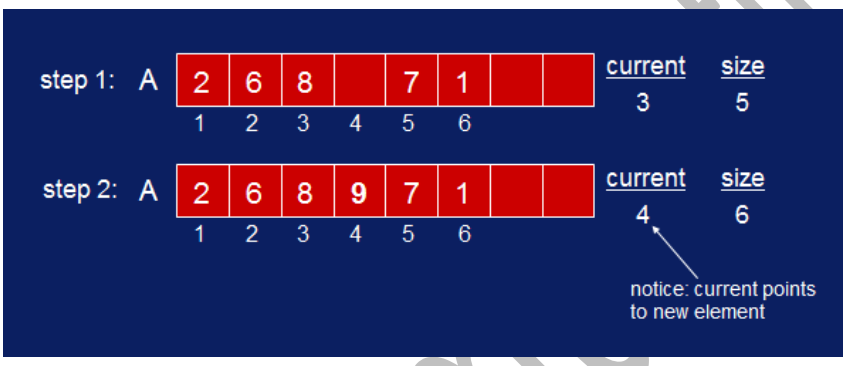
- Implementing Lists using an array: for example, the list of integers (2, 6, 8, 7, 1) could be represented as:

5	5	A	6	8	7			Current	Size
	1	1	2	3	4			3	5

elon musk files

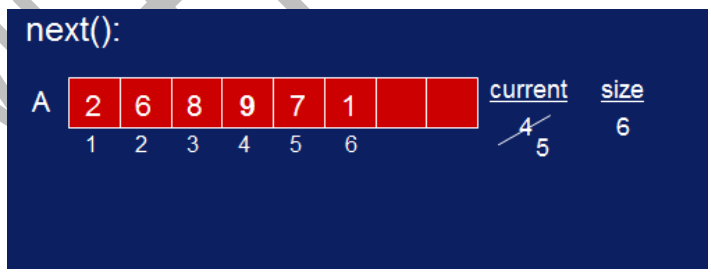
I. List Implementation by add Method

Now we will talk about adding an element to the list. Suppose there is a call to add an element in the list Add (9); current position is 3. The new list would thus be: (2, 6, 8, 9, 7, and 1)
We will need to shift everything to the right of 8 one place to the right to make place for the new element '9'.



II. List Implementation by next Method

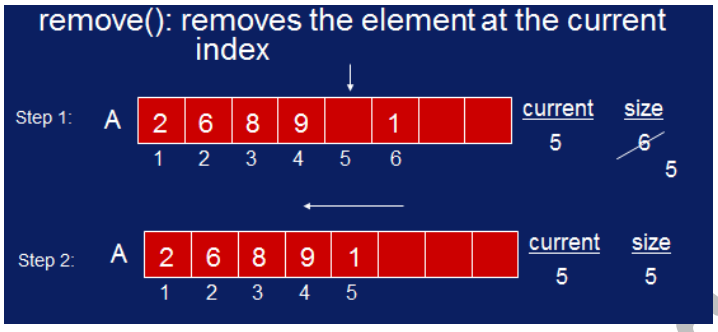
Now let's see another method, called 'next'. We have talked that the next method moves the current position one position forward. In this method, we do not add a new element to the list but simply move the pointer one element ahead.



III. List Implementation by remove Method

The removal of the element will be carried out as follows. Suppose there are 6 elements (2, 6, 8, 9, 7, and 1) in the list. The current pointer is pointing to

the position 5 that has the value 7.We fill the blank spot left by the removal of 7 by shifting the values to the right of position 5 over to the left one space.



IV. List Implementation by find Method

Now let's talk about a function, used to find a specific element in the array. The find (x) function is used to find a specific element in the array.

Find(X): traverse the array until X is located.

```
int find(int X)
{
int j;
for(j=1; j < size+1; j++ )
if( A[j] == X ) break;

if( j < size+1 )
{  // found X
current = j;        // current points to where X found
return 1;   // 1 for true
}
return 0;  // 0 (false) indicates not found
}
```

V. List Implementation by Other Methods

vu elon musk files

Other operations:

get()	→ return A[current];
update(X)	→ A[current] = X;
length()	→ return size;
back()	→ current--;
start()	→ current = 1;
end()	→ current = size;

Analysis of Array List

Now we analyze the implementation of the list while using an array internally. We analyze different methods used for the implementation of the list.

- add

  We have to move every element to the right of current to make space for the new element. Worst-case is when we insert at the beginning; we have to move every element right one place.
  Average-case: on average we may have to move half of the elements.

- Remove

  Worst-case: remove at the beginning, must shift all remaining elements to the left. Average-case: expect to move half of the elements.

- Find

  Worst-case: may have to search the entire array
  Average-case: search at most half the array.

List Using Linked Memory

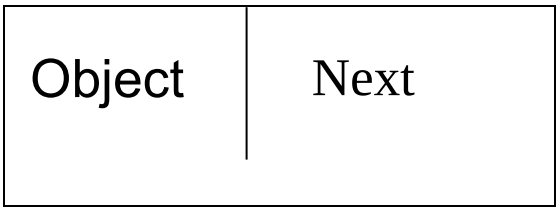
Various cells of memory are not allocated consecutively in memory. Not enough to store the elements of the list. With arrays, the second element was right next to the first element. Now the first element must explicitly tell us

vu Elon musk files

where to look for the second element. Do this by holding the memory address of the second element.

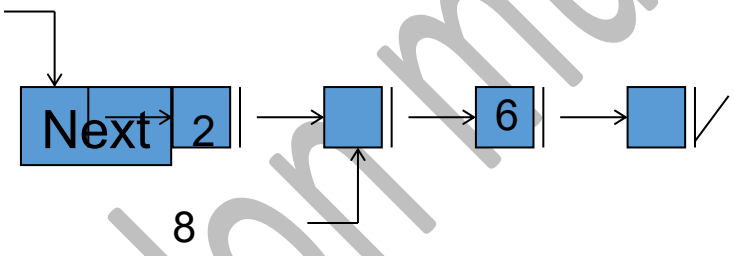
Linked List

Create a structure called a Node.



The object field will hold the actual list element. The next field in the structure will hold the starting location of the next node. Chain the nodes together to form a linked list.

List (2, 6, 7, 8, and 1) stored as a linked list:

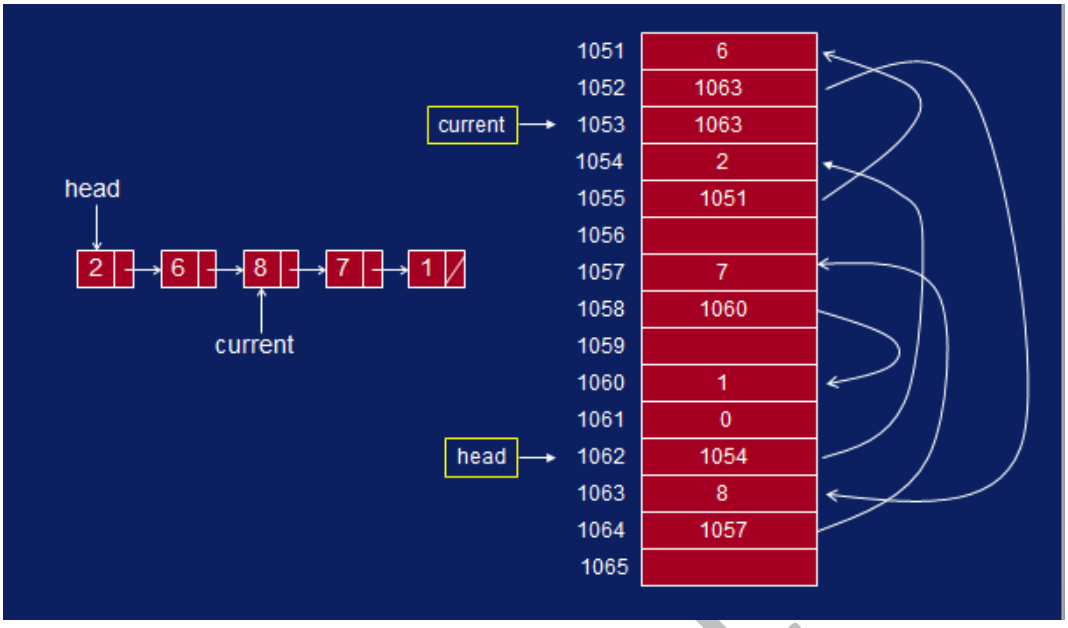


Note some features of the list:

- Need a *head* to point to the first node of the list. Otherwise we won't know where the start of the list is.
- The *current* here is a pointer, not an index.
- The next field in the last node points to *nothing*. We will place the memory address NULL which is guaranteed to be inaccessible.

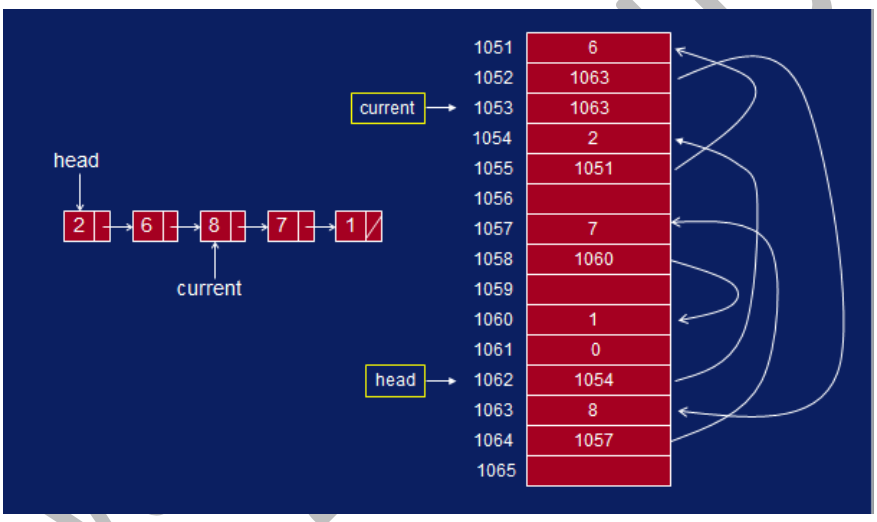
Actual picture in memory:

vu elon musk files



CS301 – Data Structures Lecture No. 03

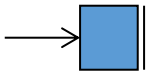
Linked List



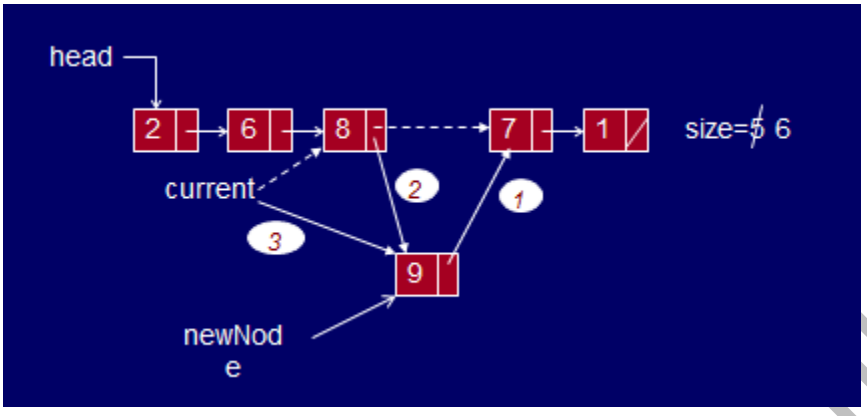
Linked List Operations

The linked list data structure provides operations to work on the nodes inside the list. The first operation we are going to discuss here is to create a new node in the memory. Add (9): Create a new node in memory to hold '9'

Node* new Node = new Node (9);



Link the new node into the list



Linked List Using C++

```
class Node {
public:
        int get() { return object; };
        void set(int object) { this->object = object; };
        Node *getNext() { return nextNode; };
        void setNext(Node *nextNode)
        { this->nextNode = nextNode; };
private:
        int object;
        Node *nextNode;
};
```

Linked List Using C++

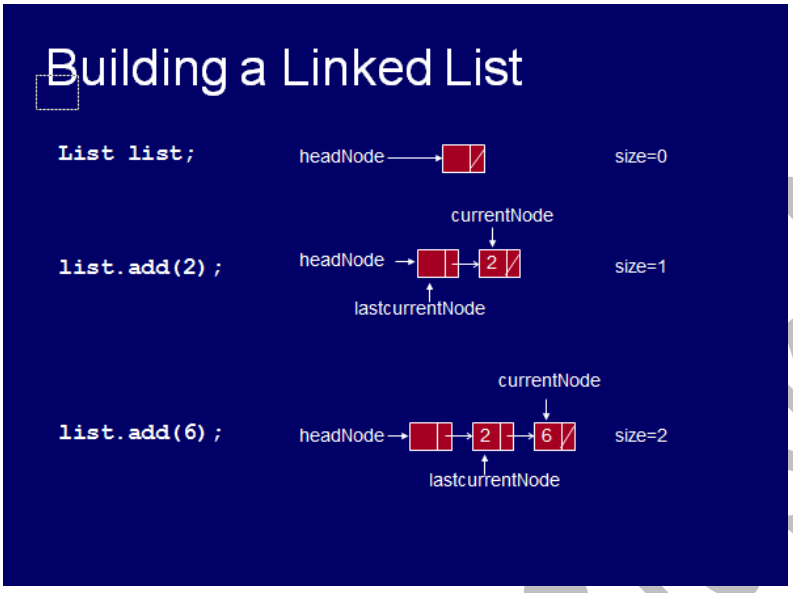
```
#include <stdlib.h>
```

vu Elon musk files

```
#include "Node.cpp"
class List {
public:
        // Constructor
        List() {
                headNode = new Node();
                headNode->setNext(NULL);
                currentNode = NULL;
                size = 0;
        };
  void add(int addObject) {
                Node* newNode = new Node();
                newNode->set(addObject);
                if( currentNode != NULL ){
            newNode->setNext(currentNode->getNext());
                   currentNode->setNext( newNode );
                   lastCurrentNode = currentNode;
                   currentNode = newNode;
      }
      else {
                   newNode->setNext(NULL);
                   headNode->setNext(newNode);
                   lastCurrentNode = headNode;
```

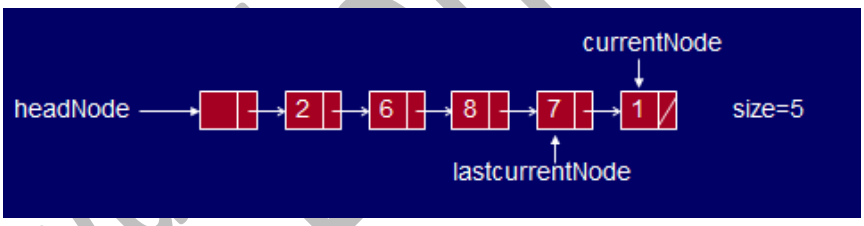
vu Elon musk files

```
        currentNode =  newNode;
    }
    size++;
};
```



Building a Linked List

List.add(8); list.add(7); list.add(1);



C++ Code for Linked List

```
int get() {
        if (currentNode != NULL)
            return currentNode->get();
};
```

```
bool next() {
        if (currentNode == NULL) return false;
        lastCurrentNode = currentNode;
        currentNode = currentNode->getNext();
        if (currentNode == NULL || size == 0)
                return false;
        else
                return true;
};
```

Data Structures Lecture No. 04

Methods of Linked List

In the previous lecture, we discussed the methods of linked list. These methods form the interface of the link list.

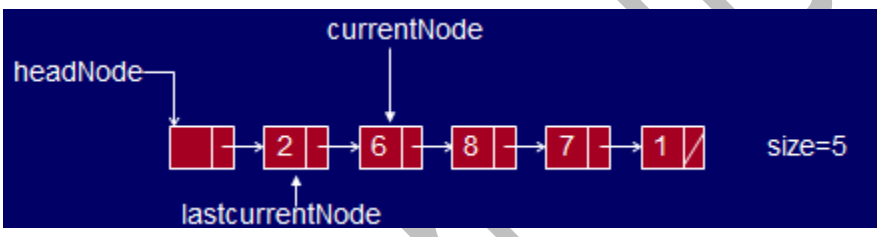
```
// position current before the first
// list element
void start() {
                lastCurrentNode = headNode;
                currentNode = headNode;
};
```

There are two statements in this method. We assign the value of head Node to both last Current Node and current Node. These two pointers point at different nodes of the list..

vu Elon musk files

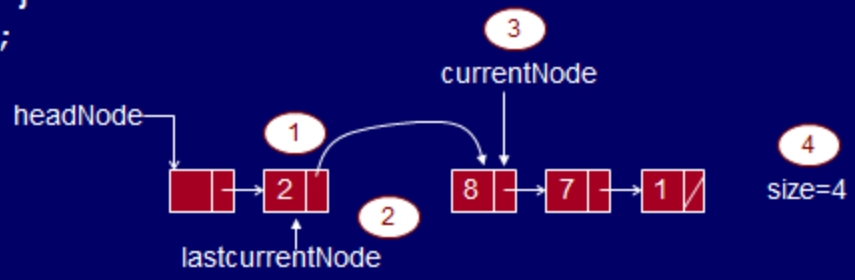
```
void remove() {
  if( currentNode != NULL &&
      currentNode != headNode) {
                lastCurrentNode->setNext(currentNode->getNext());
                delete currentNode;
                currentNode = lastCurrentNode->getNext();
                size--;
  }
};
```

We will now see how a node can be removed from the link list. We use the method remove for this purpose.



vu Elon Musk files

```
void remove() {
  if( currentNode != NULL &&
      currentNode != headNode) {
    1 lastCurrentNode->setNext(currentNode->getNext());
    2 delete currentNode;
    3 currentNode = lastCurrentNode->getNext();
    4 size--;
  }
};
```



C++ Code for Linked List

int length()

{

return size;

};

private:

int size;

Node *headNode;

Node *currentNode, *lastCurrentNode;

Example of list usage

#include <iostream>

#include <stdlib.h>

vu Elon musk

```
#include "List.cpp"

int main(int argc, char *argv[])

{

        List list;

        list.add(5); list.add(13); list.add(4);

        list.add(8); list.add(24); list.add(48); list.add(12);

        list.start();

        while (list.next())

                cout << "List Element: "<< list.get()<<endl;

}
```

Analysis of Linked List

We will see whether it is useful or not. We will see its cost and benefit with respect to time and memory.

- add

  We simply insert the new node after the current node. So add is a one-step operation.

- remove

  Remove is also a one-step operation

- find

  Worst-case: may have to search the entire list

- back

  Moving the current pointer back one node requires traversing the list from the start until the node whose next pointer points to current node.

Doubly-linked List

vu elon musk files

Moving forward in a singly-linked list is easy; moving backwards is not so easy. To move back one node, we have to start at the head of the singly-linked list and move forward until the node before the current. To avoid this we can use *two* pointers in a node: one to point to next node and another to point to the previous node:

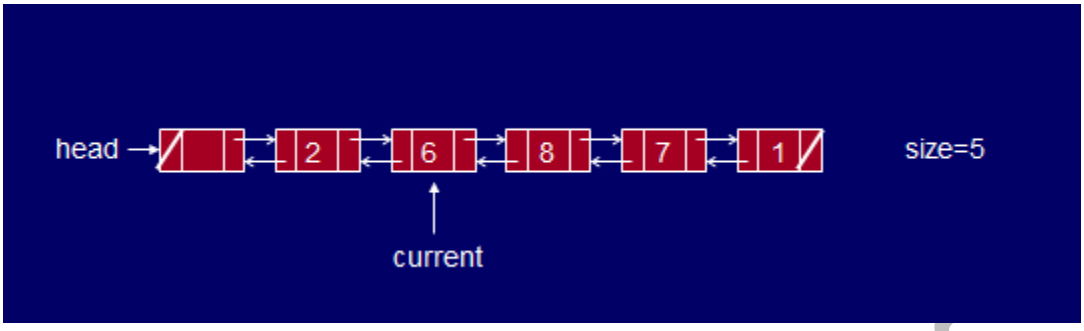
prev	element	next

```
class Node {
public:
        int get() { return object; };
        void set(int object) { this->object = object; };
        Node* getNext() { return nextNode; };
        void setNext(Node* nextNode)
        { this->nextNode = nextNode; };
        Node* getPrev() { return prevNode; };
        void setPrev(Node* prevNode)
        { this->prevNode = prevNode; };
private:
        int object;
        Node* nextNode;
        Node* prevNode;
};
```

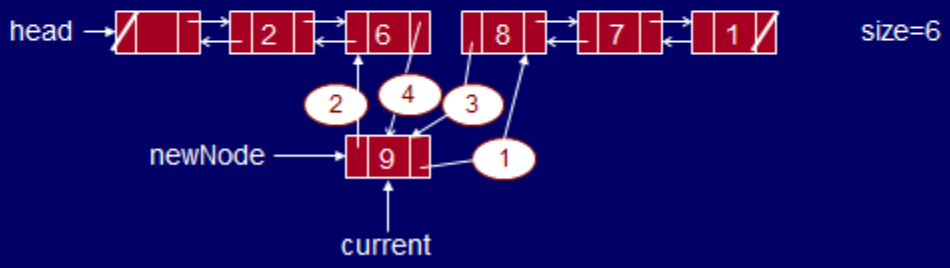
- Need to be more careful when adding or removing a node.

vu elon musk files

- Consider add: the order in which pointers are reorganized is important:



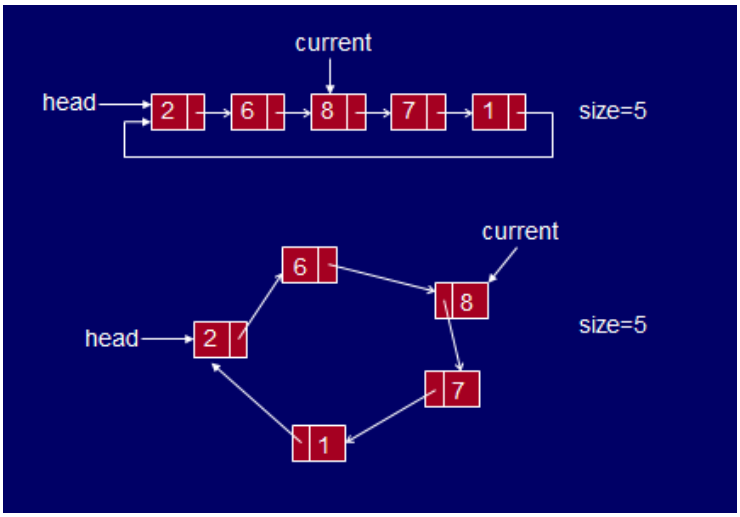

```
1.   newNode->setNext( current->getNext() );
2.   newNode->setprev( current );
3.   (current->getNext())->setPrev(newNode);
4.   current->setNext( newNode );
5.   current = newNode;
6.   size++;
```



Circularly-linked lists

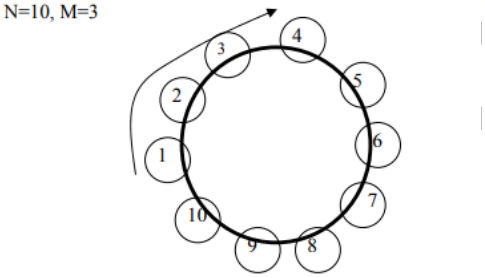
The next field in the last node in a singly-linked list is set to NULL. Moving along a singly-linked list has to be done in a watchful manner. Doubly-linked lists have two NULL pointers: prev in the first node and next in the last node. A way around this potential hazard is to link the last node with the first node in the list to create a *circularly-linked list.*

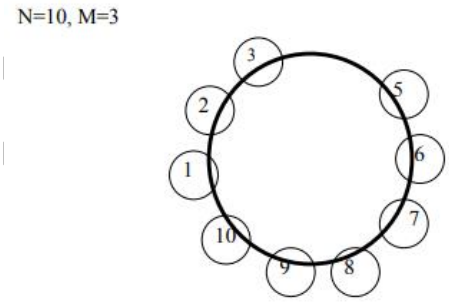
- Two views of a circularly linked list:



Josephus Problem

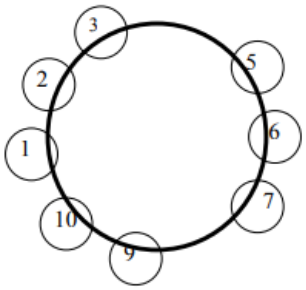
A case where circularly linked list comes in handy is the solution of the *Josephus Problem*. Consider there are 10 persons. They would like to choose a leader. The way they decide is that all 10 sit in a circle. They start a count with person 1 and go in clockwise direction and skip 3. Person 4 reached is eliminated. The count starts with the fifth and the next person to go is the fourth in count. Eventually, a single person remains.







N=10, M=3

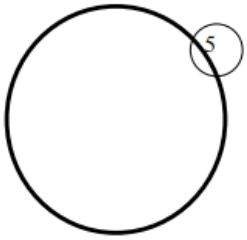


Eliminated

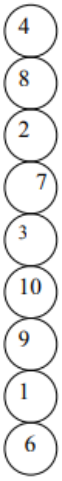


In the end, only number five will remain intact.

N=10, M=3



Eliminated



Example

```
#include "CList.cpp"
void main(int argc, char *argv[])
{
        CList list;
        int i, N=10, M=3;
        for(i=1; i <= N; i++ ) list.add(i);
        list.start();
        while( list.length() > 1 ) {
```

```
        for(i=1; i <= M; i++ ) list.next();
    cout << "remove: " << list.get() << endl;
    list.remove();
      }
      cout << "leader is: " << list.get() << endl;
}
```

Data Structures Lecture No. 05

Josephus Problem

```
#include "CList.cpp"
void main(int argc, char *argv[])
{
      CList list;
      int i, N=10, M=3;
      for(i=1; i <= N; i++ ) list.add(i);
      list.start();
      while( list.length() > 1 ) {
        for(i=1; i <= M; i++ ) list.next();
    cout << "remove: " << list.get() << endl;
    list.remove();
      }
      cout << "leader is: " << list.get() << endl;
}
```

vu Elon musk files

Using a circularly-linked list made the solution trivial. The solution would have been more difficult if an array had been used. This illustrates the fact that the choice of the appropriate data structures can significantly simplify an algorithm. It can make the algorithm much faster and efficient. Later we will see how some elegant data structures lie at the heart of major algorithms. An entire CS course "Design and Analysis of Algorithms" is devoted to this topic.

Benefits of using circular list

Some problems are circular and a circular data structure would be more natural when used to represent it. The entire list can be traversed starting from any node (traverse means visit every node just once) **fewer special cases when coding** (all nodes have a node before and after it).

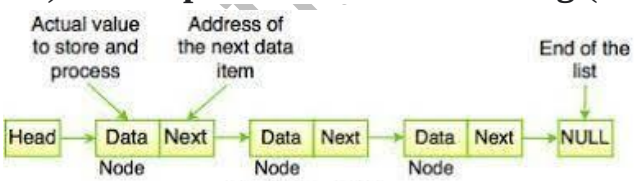


Fig. Linked List

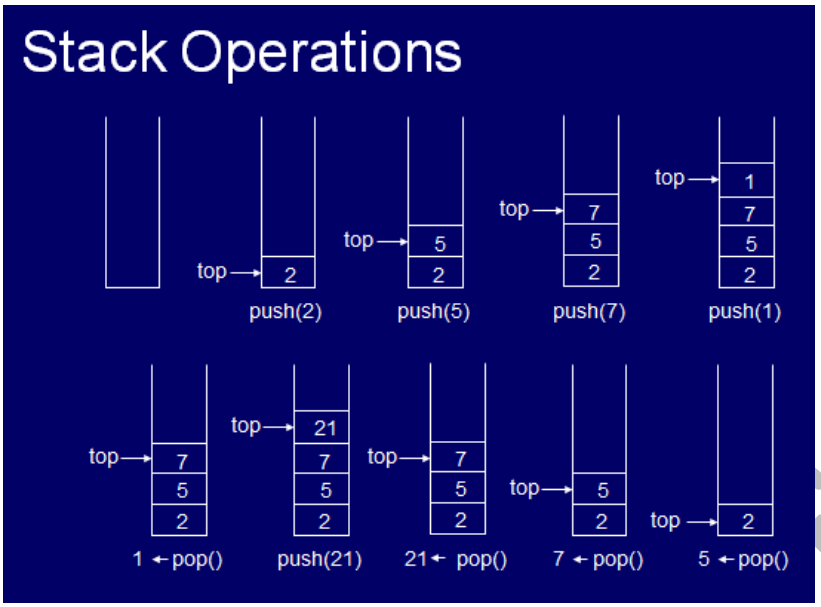
Abstract Data Type

Abstract Data type (ADT) is a **type (or class) for objects whose behavior is defined by a set of value and a set of operations** It is called "abstract" because it gives an implementation-independent view. The process of providing only the essentials and hiding the details is known as abstraction.

Stacks

A stack is **an abstract data type that holds an ordered, linear sequence of items**. In contrast to a queue, a stack is a last in, first out (LIFO) structure. A real-life example is a stack of plates: you can only take a plate from the top of the stack, and you can only add a plate to the top of the stack.

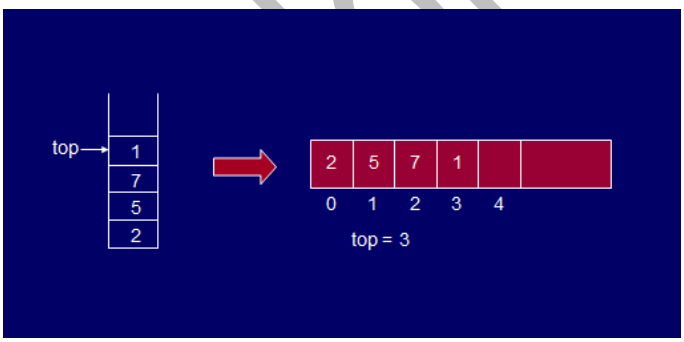
- Push(X) – insert X as the top element of the stack
- Pop() – remove the top element of the stack and return it.
- Top() – return the top element without removing it from the stack.



The last element to go into the stack is the first to come out: *LIFO* – Last in First Out. Have Is Empty () Boolean function that returns true if stack is empty, false otherwise. Throw Stack Empty exception: advanced C++ concept.

Stack Implementation using array

Worst case for insertion and deletion from an array when insert and delete from the beginning: shift elements to the left. Best case for insert and delete is at the end of the array – no need to shift any elements. Implement push () and pop () by inserting and deleting at the end of an array.



In case of an array, it is possible that the array may "fill-up" if we push enough elements. Have a Boolean function **Is Full ()** which returns true is stack (array) is full, false otherwise. We would call this function before calling push(x).

elon musk files

```
int pop()
{
   return A[current--];
}
void push(int x)
{
   A[++current] = x;
}
int top()
{
   return A[current];
}
int IsEmpty()
{
   return ( current == -1 );
}
int IsFull()
{
   return ( current == size-1);
}
```

A quick examination shows that all five operations take constant time.

Data Structures Lecture No. 06

vu Elon musk files

Stack From the Previous Lecture

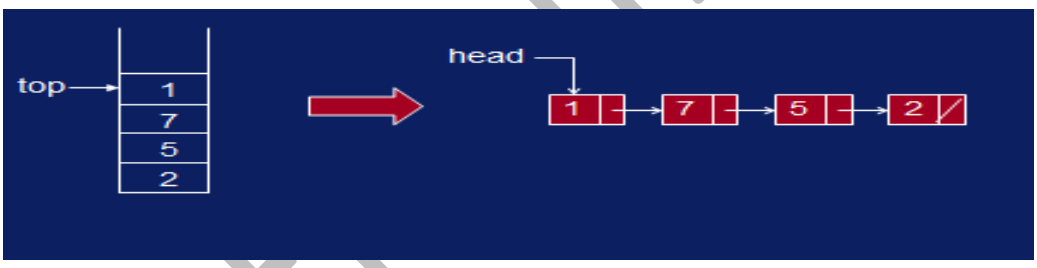
Is Empty () method is implemented as a stack can be empty like a list or set structures. It is important to understand that is Full () method is there in stack implementation because of limitation of array but is Empty () method is part of the stack characteristics or functionality.

Stack Using Linked List

We can avoid the size limitation of a stack implemented with an array by using a linked list to hold the stack elements. As with array, however, we need to decide where to insert elements in the list and where to delete them so that push and pop will run the fastest.

For a singly-linked list, insert at start or end takes constant time using the head and current pointers respectively. Removing an element at the start is constant time but removal at the end required traversing the list to the node one before the last. Make sense to place stack elements at the start of the list because insert and removal are constant time.

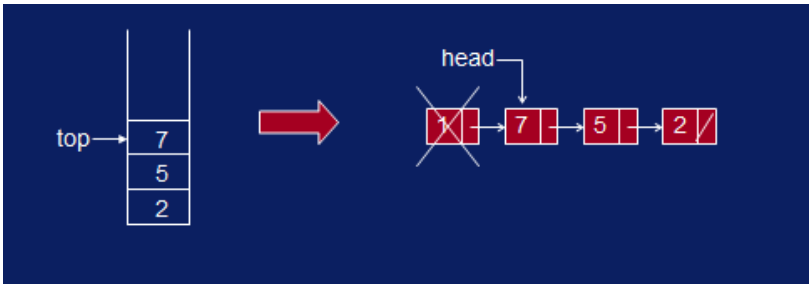
- No need for the current pointer; head is enough.



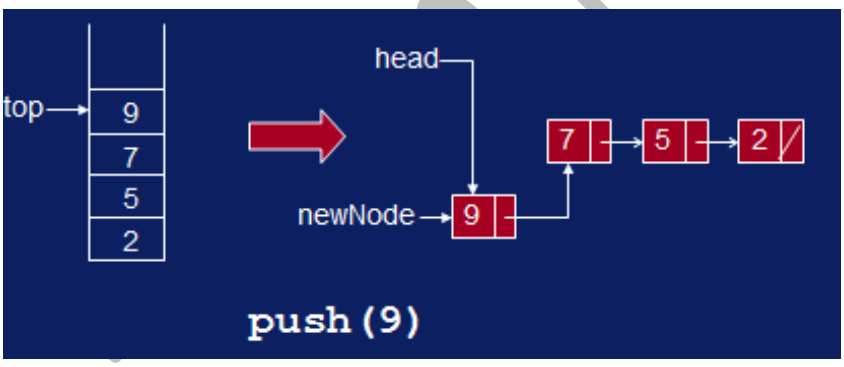

```
int pop()
{
    int x = head->get();
    Node* p = head;
    head = head->getNext();
    delete p;
```

vu Exams musk files

```
    return x;
}
```




```
void push(int x)
{
    Node* newNode = new Node();
    newNode->set(x);
    newNode->setNext(head);
    head = newNode;
}
```




```
int top()
{
    return head->get();
}
int IsEmpty()
```

musk files

```
{
   return ( head == NULL );
}
```

- All four operations take constant time.

Stack Implementation: Array or Linked List

Allocating and deal locating memory for list nodes does take more time than reallocated array. List uses only as much memory as required by the nodes; array requires allocation ahead of time. List pointers (head, next) require extra memory. Array has an upper limit; List is limited by dynamic memory allocation.

Use of Stack

Example of use: prefix, infix, postfix expressions. Consider the expression A+B: we think of applying the *operator* "+" to the *operands* A and B. "+" are termed a *binary operator*: it takes two operands. Writing the sum as A+B is called the *infix* form of the expression.

Prefix, Infix, Postfix

Two other ways of writing the expression are

+ A B *prefixes*

A B + *postfix*

The prefixes "pre" and "post" refer to the position of the operator with respect to the two operands.

Consider the infix expression

A + B * C

We "know" that multiplication is done before addition.

The expression is interpreted as

A + (B * C)

Multiplication has *precedence* over addition.

I. Conversion to postfix

vu Elon musk files

A + (B * C) infix form

A + (B C *) convert multiplication

A (B C *) + convert addition

A B C * + postfix form

II. Conversion to postfix

(A + B) * C infix form

(A B +) * C convert addition

(A B +) C * convert multiplication

A B + C * postfix form

Precedence of Operators

There are five binary operators, called addition, subtraction, multiplication, division and exponentiation. The order of precedence is (highest to lowest)

- Exponentiation ↑
- Multiplication/division *, /
- Addition/subtraction +, -

For operators of same precedence, the left-to-right rule applies:

A+B+C means (A+B) +C.

For exponentiation, the right-to-left rule applies

A ↑ B ↑ C means A ↑ (B ↑ C)

Infix to Postfix

vu Elon musk files

Infix	Postfix
A + B	A B +
12 + 60 – 23	12 60 + 23 –
(A + B)*(C – D)	A B + C D – *
A ↑ B * C – D + E/F	A B ↑ C*D – E F/+

Data Structures Lecture No. 07

Evaluating postfix expressions

Infix	Postfix
A + B	A B +
12 + 60 – 23	12 60 + 23 –
(A + B)*(C – D)	A B + C D – *
A ↑ B * C – D + E/F	A B ↑ C*D – E F/+

Note that the postfix form an expression does not require parenthesis Consider '4+3*5' and '(4+3)*5'. The parentheses are not needed in the first but they are necessary in the second.

The postfix forms are:

4+3*5 435*+

(4+3)*5 43+5*

Each operator in a postfix expression refers to the previous two operands. Each time we read an operand, we push it on a stack. When we reach an operator, we pop the two operands from the top of the stack, apply the operator and push the result back on the stack.

You will understand the algorithm

Stack s; // declare a stack

while(not end of input) // not end of postfix expression

vu Elon musk files

```
{
        e = get next element of input
        if( e is an operand )
                s.push( e );
        else {
                op2 = s.pop();
                op1 = s.pop();
                value = result of applying operator 'e' to op1 and op2;
                s.push( value );
        }
}
finalresult = s.pop();
```

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

An example Evaluating Postfix

Input	op1	op2	value	stack
6				**6**
2				**6,2**
3				**6,2,3**
+	**2**	**3**	**5**	**6,5**
-	**6**	**5**	**1**	**1**
3	**6**	**5**	**1**	**1,3**
8	**6**	**5**	**1**	**1,3,8**

vu Elon musk files

2	**6**	**5**	**1**	**1,3,8,2**
/	**8**	**2**	**4**	**1,3,4**
+	**3**	**4**	**7**	**1,7**
*****	**1**	**7**	**7**	**7**
2	**1**	**7**	**7**	**7,2**
↑	**7**	**2**	**49**	**49**
3	**7**	**2**	**49**	**49,3**
+	**49**	**3**	**52**	**52**

Converting Infix to Postfix

Consider the infix expressions 'A+B*C' and '(A+B)*C'. The postfix versions are 'ABC*+' and 'AB+C*'. The order of operands in postfix is the same as the infix. In scanning from left to right, the operand 'A' can be inserted into postfix expression. The '+' cannot be inserted until its second operand has been scanned and inserted. The '+' has to be stored away until its proper position is found. When 'B' is seen, it is immediately inserted into the postfix expression. Can the '+' be inserted now? In the case of 'A+B*C' cannot because * has precedence. In case of '(A+B)*C', the closing parenthesis indicates that '+' must be performed first. Assume the existence of a function 'prcd(op1,op2)' where op1 and op2 are two operators. Prcd(op1,op2) returns TRUE if op1 has precedence over op2, FASLE otherwise.

- prcd('*','+') is TRUE
- prcd('+','+') is TRUE
- prcd('+','*') is FALSE

Here is the algorithm that converts infix expression to its postfix form.The infix expression is without parenthesis.

1. **Stack s;**

vu elon musk files

```
2.  While( not end of input ) {
3.      c = next input character;
4.      if( c is an operand )
5.          add c to postfix string;
6.      else {
7.          while( !s.empty() && prcd(s.top(),c) ){
8.              op = s.pop();
9.              add op to the postfix string;
10.         }
11.         s.push( c );
12.     }
13.     while( !s.empty() ) {
14.         op = s.pop();
15.         add op to postfix string;
16.     }
```

Example: A + B * C

symb	postfix	stack
A	A	
+	A	+
B	AB	+
*	AB	+ *
C	ABC	+ *

vu elon musk files

ABC * +

ABC * +

Handling parenthesis important for Quiz

When an open parenthesis '(' is read, it must be pushed on the stack. This can be done by setting prcd(op,'(') to be FALSE. Also, prcd('(',op) == FALSE which ensures that an operator after '(' is pushed on the stack. When a ')' is read, all operators up to the first '(' must be popped and placed in the postfix string. To do this, prcd(op,')') == TRUE. Both the '(' and the ')' must be discarded: prcd('(',')') == FALSE.

Need to change line 11 of the algorithm.

We have to change the line s.push(c) to:

```
if( s.empty()  ||  symb != ')' )
s.push( c );

else
s.pop(); // discard the '('
```

- prcd('(', op) = FALSE for any operator
- prcd(op, '(') = FALSE for any operator other than '('
- prcd(op, ')') = TRUE for any operator other than '('
- prcd(')', op) = error for any operator.

Data Structures Lecture No. 08

Conversion from infix to postfix

Example: (A + B) * C

symb	postfix	stack
(		(

vu Elon musk files

A	A	(
+	A	(+
B	AB	(+
)	AB +	
*	AB +	*
C	AB + C	*
	AB + C *	

C++ Templates

We need a stack of operands and a stack of operators. Operands can be integers and floating point numbers, even variables. Operators are single characters. We would have to create classes Float Stack and Char Stack. Yet the internal workings of both classes are the same.

We can use C++ Templates to create a "template" of a stack class. Instantiate float stack, char stack, or stack for any type of element we want.

This is written in the file Stack.h.

Stack.h:

```
template <class T>
class Stack {
public:
   Stack();
   int empty(void); // 1=true, 0=false
   int push(T &); // 1=successful,0=stack overflow
   T pop(void);
   T peek(void);
```

vu Elon musk files

```
    ~Stack();
private:
    int top;
    T* nodes;
};
```

Implementation

Now we will see how to implement this stack class in our program. We save this code in the file named Stack.cpp.

Stack.cpp

```
#include <iostream.h>
#include <stdlib.h>
#include "Stack.cpp"
#define MAXSTACKSIZE 50
template <class T>
Stack<T>::Stack()
{
    top = -1;
    nodes = new T[MAXSTACKSIZE];
}
template <class T>
Stack<T>::~Stack()
{
    delete nodes;
```

vu elon musk files

```
}
template <class T>
int Stack<T>::empty(void)
{
   if( top < 0 ) return 1;
   return 0;
}
template <class T>
T Stack<T>::pop(void)
{
   T x;
   if( !empty() ) {
      x = nodes[top--];
      return x;
   }
   cout << "stack underflow in pop.\n";
   return x;
}
```

main.cpp

```
#include "Stack.cpp"
int main(int argc, char *argv[]) {
  Stack<int> intstack;
  Stack<char> charstack;
```

vu Elon musk files

```
  int x=10, y=20;
  char c='C', d='D';

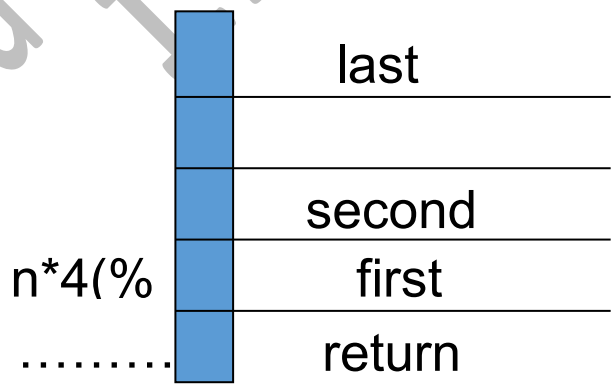
  intstack.push(x);   intstack.push(y);
  cout << "intstack: " << intstack.pop() << ", "
            << intstack.pop() << "\n";

  charstack.push(c); charstack.push(d);
  cout << "charstack: " << charstack.pop() << ", "
      << charstack.pop() << "\n";
}
```

Function Call Stack

Stacks play a key role in implementation of function calls in programming languages. In C++, for example, the "call stack" is used to pass function arguments and receive return values. The call stack is also used for "local variables"

In GCC, a popular C/C++ compiler on Intel platform, stack entries are:



vu flon musk files

Example: consider the function:

```
 int i_avg (int a, int b)
 {
   return (a + b) / 2;
 }
# Stack layout on entry:
#
# 8(%esp) b
# 4(%esp) a
#  (%esp) return address
```

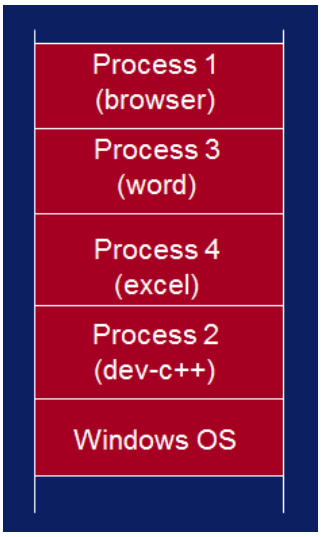
Example: consider the function:

```
 int i_avg (int a, int b)
 {
   return (a + b) / 2;
 }
.globl _i_avg
_i_avg:
   movl 4(%esp), %eax
   addl 8(%esp), %eax # Add the args
   sarl $1, %eax # Divide by 2
   ret # Return value is in %eax
```

Memory Organization

www.elonmusk.com

When a program (.exe) is run, it is loaded in memory. It becomes a *process*. The process is given a block of memory. [Control-Alt-DEL]



v files

Windows Task Manager

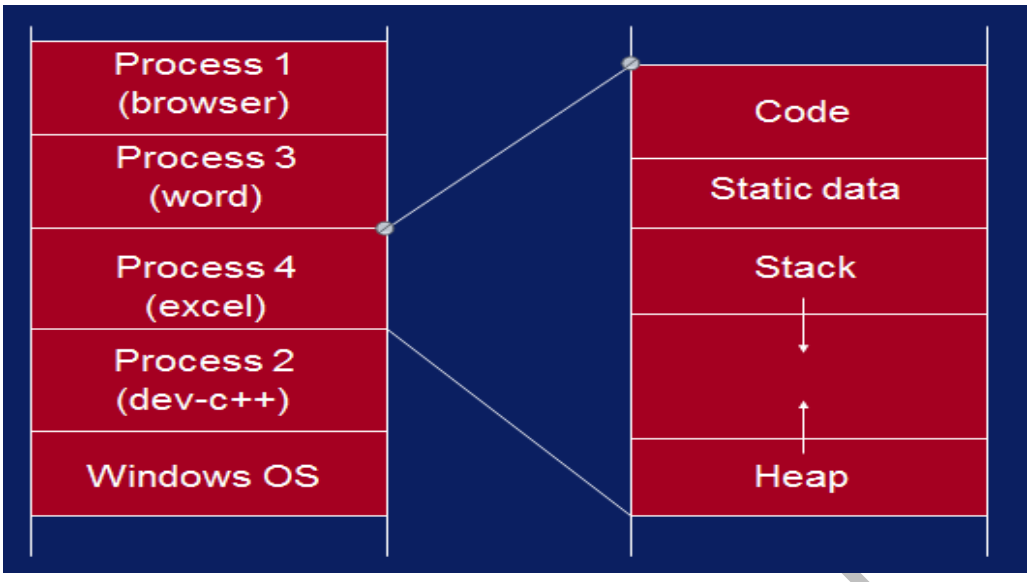
File Options View Help

Applications | Processes | Performance

Image Name	PID	CPU	CPU Time	Mem Usage	Page Faults	I/O Reads	I/O Writes	I/O Read Bytes	I/O Write Bytes
svchost.exe	468	00	0:00:02	8,048 K	2,413	41,100	39,264	101,443,489	100,402,717
SERVICES.EXE	224	00	0:00:03	5,788 K	2,056	13,009	11,765	1,444,964	914,388
POProxy.exe	1088	00	0:00:01	5,428 K	1,453	1,860	0	883,006	0
alertsvc.exe	844	00	0:00:00	4,764 K	1,338	42	2	13,566	12
explorer.exe	888	00	0:00:20	4,696 K	16,182	8,744	127	7,060,760	2,358,902
SPOOLSV.EXE	428	00	0:00:00	3,640 K	1,336	2,095	2	1,008,896	12
svchost.exe	400	00	0:00:00	3,532 K	3,809	74	5	32,104	288
navapsvc.exe	520	00	0:00:01	3,244 K	2,960	139,462	2	6,069,556	12
mstask.exe	632	00	0:00:00	3,116 K	859	48	15	19,436	356
POWERPNT.EXE	984	00	0:00:13	2,428 K	8,602	29,869	88	2,696,072	447,454
taskmgr.exe	836	01	0:00:02	2,424 K	617	0	0	0	0
CSRSS.EXE	172	00	0:00:15	1,956 K	3,037	39,254	0	981,959	0
navapw32.exe	1172	00	0:00:00	1,752 K	484	99	0	53,446	0
npssvc.exe	580	00	0:00:00	1,644 K	410	19	4	2,128	72
hidserv.exe	488	00	0:00:00	1,352 K	338	2	2	40	12
LSASS.EXE	236	00	0:00:00	868 K	2,624	1,249	807	1,469,654	137,653
regsvc.exe	616	00	0:00:00	740 K	188	2	2	54	12
WINLOGON.EXE	192	00	0:00:06	544 K	5,381	238	70	2,860,452	35,333
SMSS.EXE	144	00	0:00:00	336 K	621	236	40	14,985,728	31,744
System	8	00	0:00:27	212 K	3,160	71	1,749	33,616	1,387,471
WinMgmt.exe	684	00	0:00:10	164 K	5,911	464	23,934	5,308,962	6,611,687
System Idle Process	0	99	1:19:22	16 K	1	0	0	0	0

End Process

Processes: 22 | CPU Usage: 1% | Mem Usage: 88124K / 436532K

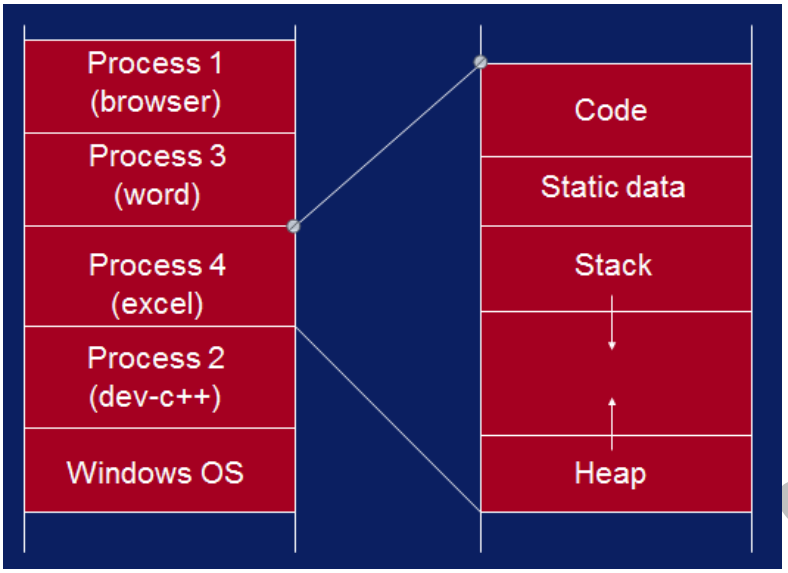


Data Structures Lecture No. 09

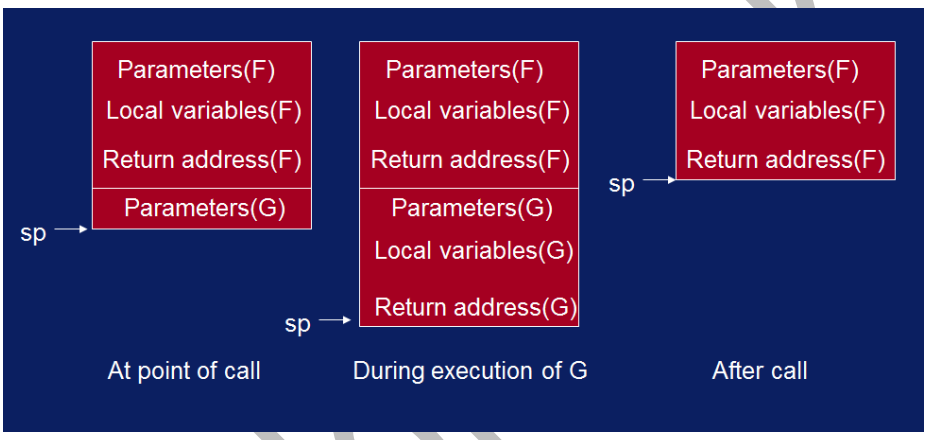
Memory Organization

The memory is organized in the **form of a cell**; each cell is able to be identified with a unique number called address. Each cell is able to recognize control signals such as “read” and “write”, generated by CPU when it wants to read or write address. To access the instruction CPU generates the memory request.

- A code section that contains the binary version of the actual code of the program written in some language like C/C++
- A section for static data including global variables
- A stack and – Finally
- a heap



Here is stack layout when function F calls function G:



Queues

A queue is a FIFO (First In First Out — the element placed at first can be accessed at first) structure which can be commonly found in many programming languages. This structure is named as "queue" because it resembles a real-world queue — people waiting in a queue. Queue operations given below are the 2 basic operations that can be performed on a queue.

Enqueue(X) – place X at the *rear* of the queue.

Dequeue() --remove the *front* element and return it.

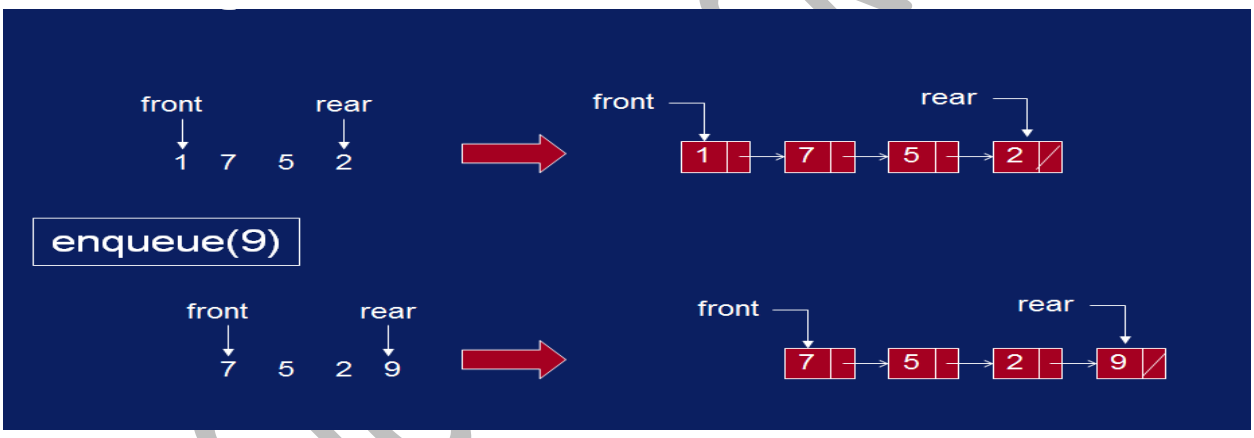
Front() -- return front element without removing it.

IsEmpty()-- return TRUE if queue is , FALSE otherwise

Implementing Queue

- Using linked List: *Recall*
- Insert works in constant time for either end of a linked list.
- Remove works in constant time only.
- Seems best that head of the linked list be the front of the queue so that all removes will be from the front.
- Inserts will be at the end of the list.

Using linked List:



int dequeue()

{

int x = front->get();

Node* p = front;

front = front->getNext();

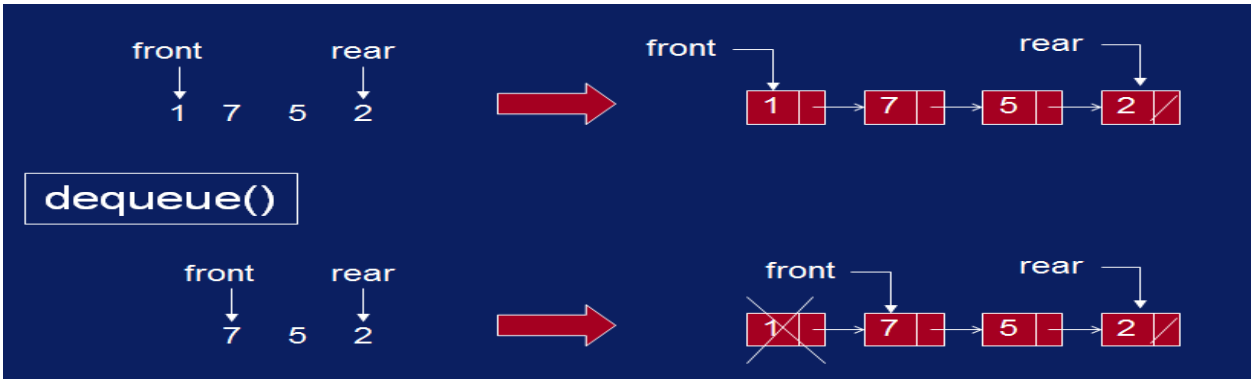
delete p;

return x;

```
}
void enqueue(int x)
{
   Node* newNode = new Node();
   newNode->set(x);
   newNode->setNext(NULL);
        rear->setNext(newNode);
   rear = newNode;
}
int front()
{
   return front->get();
}
int isEmpty()
{
            return ( front == NULL );
}
```

If we use an array to hold queue elements, both insertions and removal at the front (start) of the array are expensive. This is because we may have to shift up to "n" elements. For the stack, we needed only one end; for queue we need both. To get around this, we will not shift upon removal of an element.

vu Elon musk files




```
int dequeue()
{
    int x = front->get();
    Node* p = front;
    front = front->getNext();
    delete p;
    return x;
}
void enqueue(int x)
{
    Node* newNode = new Node();
    newNode->set(x);
    newNode->setNext(NULL);
          rear->setNext(newNode);
    rear = newNode;
}
int front()
```

vu elon musk files

```
{
   return front->get();
}
int isEmpty()
{
            return ( front == NULL );
}
```

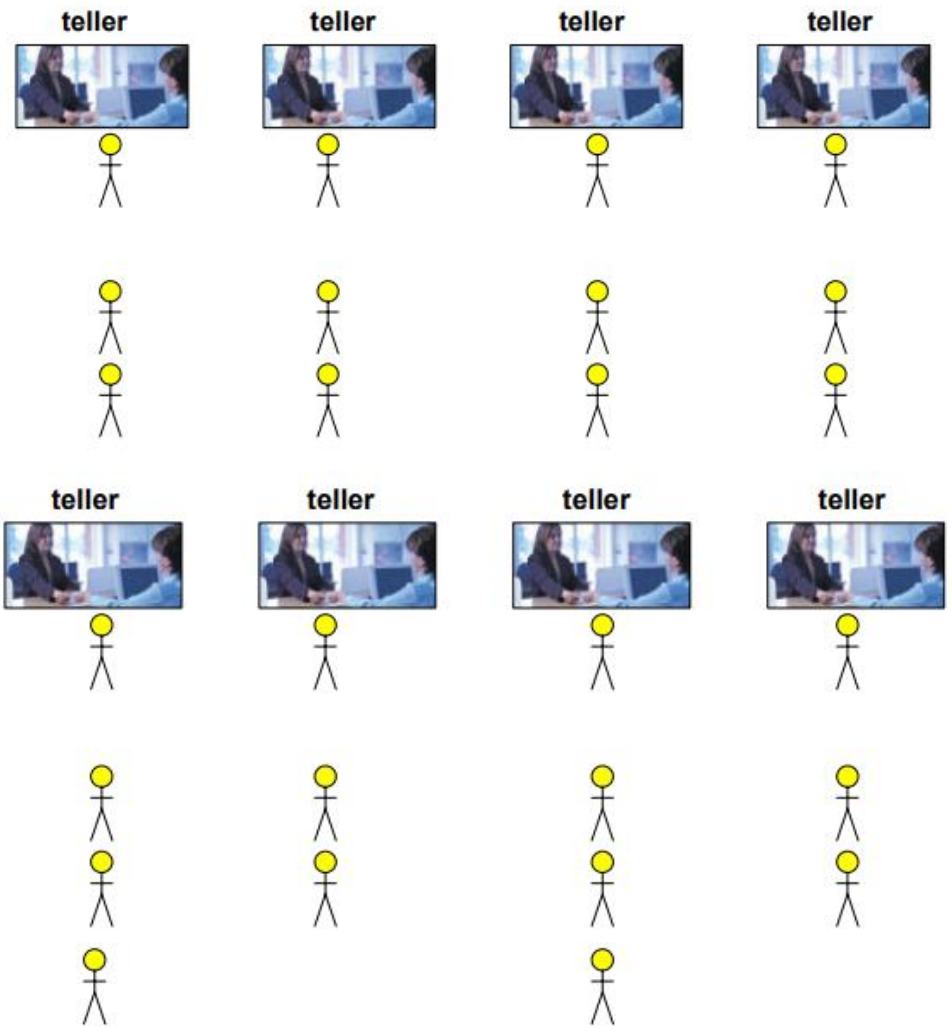
If we use an array to hold queue elements, both insertions and removal at the front (start) of the array are expensive. This is because we may have to shift up to "n" elements. For the stack, we needed only one end; for queue we need both. To get around this, we will not shift upon removal of an element.

vu Elon musk files

Data Structures Lecture No. 10

Queues

A customer enters the bank at a specific time (t_1) desiring to conduct a transaction. Any one of the four tellers can attend to the customer. The transaction (withdraws, deposit) will take a certain period of time (t_2). If a teller is free, the teller can process the customer's transaction immediately and the customer leaves the bank at t_1+t_2. It is possible that none of the four tellers is free in which case there is a line of customers are *each* teller. An arriving customer proceeds to the back of the shortest line and waits for his turn. The customer leaves the bank at t_2 time units after reaching the front of the line. The time spent at the bank is t_2 plus time waiting in line.



Simulation Models

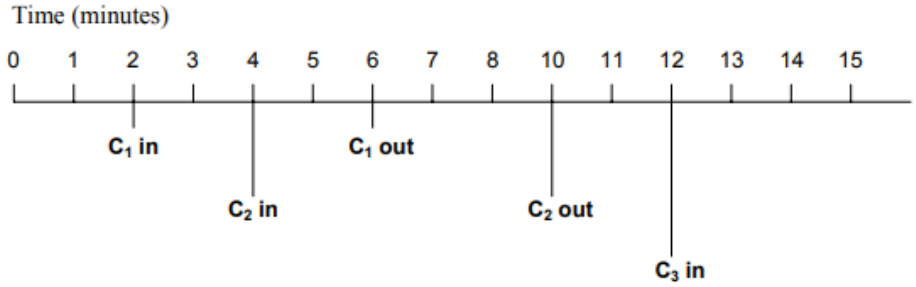
Two common models of simulation are time-based simulation and event-based simulation. In time-based simulation, we maintain a timeline or a clock. The clock ticks and things happen when the time reaches the moment of an event.

Timeline based Simulation

Consider the bank example. All tellers are free.

Customer C_1 comes in at time 2 minutes after bank opens. His transaction (withdraw money) will require 4 minutes. Customer C_2 arrives 4 minutes after the bank opens. Will need 6 minutes for transaction. Customer C3 arrives 12 minutes after the bank opens and needs 10 minutes.

We have a time line and marks for every min.



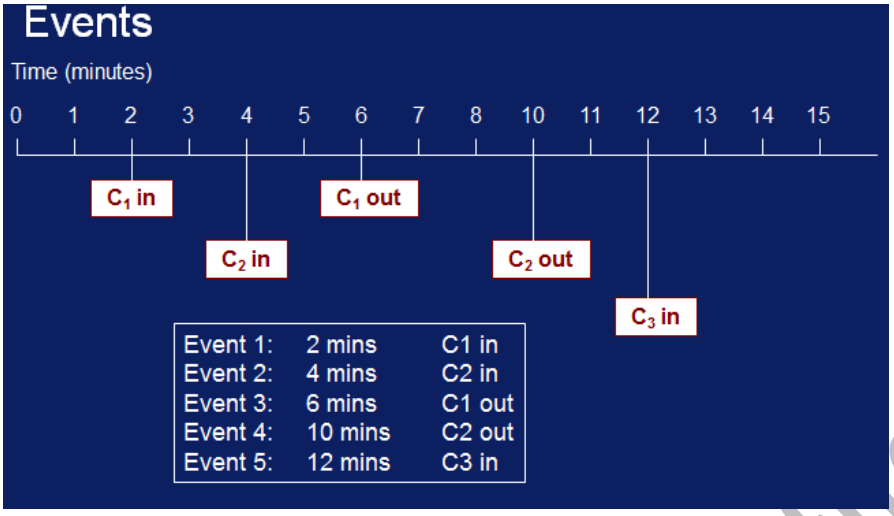
We could write a main clock loop as follows:

```
clock = 0;
while( clock <= 24*60 ) { // one day
    read new customer;
    if customer.arrivaltime == clock
     insert into shortest queue;
    check the customer at head of all four queues.
    if transaction is over, remove from queue.
    clock = clock + 1;
}
```

Event based Simulation

Don't wait for the clock to tic until the next event. Compute the time of next event and maintain a list of events in increasing order of time. Remove an event from the list in a loop and process it.

vu elon musk files



Maintain a queue of events. Remove the event with the earliest time from the queue and process it. As new events are created, insert them in the queue. A queue where the DE queue operation depends not on FIFO is called a *priority queue*.

Development of the C++ code to carry out the simulation. We will need the queue data structure. We will need the priority queue. Information about arriving customers will be placed in an input file. Each line of the file contains the items (arrival time, transaction duration).

Arriving Customers' File

Here are a few lines from the input file.

00 30 10 <- customer 1

00 35 05 <- customer 2

00 40 08

00 45 02

00 50 05

00 55 12

01 00 13

01 01 09

vu Elon musk files

"00 30 10" means Customer 1 arrives 30 minutes after bank opens and will need 10 minutes for his transaction. "01 01 09" means customer arrives one hour and one minute after bank opens and transaction will take 9 minutes.

Simulation Procedure

The first event to occur is the arrival of the first customer. This event placed in the priority queue. Initially, the four teller queues are empty.

The simulation proceeds are follows:

- When an arrival event is removed from the priority queue, a node representing the customer is placed on the shortest teller queue.
- If that customer is the only one on a teller queue, a event for his departure is placed on the priority queue.
- At the same time, the next input line is read and an arrival event is placed in the priority queue.
- When a departure event is removed from the event priority queue, the customer node is removed from the teller queue.
- The total time spent by the customer is computed: it is the time spent in the queue waiting and the time taken for the transaction.
- This time is added to the total time spent by all customers.
- At the end of the simulation, this total time divided by the total customers served will be average time spent by customers.
- The next customer in the queue is now served by the teller.
- A departure event is placed on the event queue.

Code for Simulation

```
#include <iostream>

#include <string>

#include <strstream.h>
```

vu Elon musk files

```
#include "Customer.cpp"
#include "Queue.h"
#include "PriorityQueue.cpp"
#include "Event.cpp"
Queue q[4];       // teller queues
PriorityQueue pq; //eventList;
int totalTime;
int count = 0;
int customerNo = 0;
main (int argc, char *argv[])
{
        Customer* c;
        Event* nextEvent;
        // open customer arrival file
  ifstream data("customer.dat", ios::in);
        // initialize with the first arriving
// customer.
  readNewCustomer(data);
        while( pq.length() > 0 )
   {
                nextEvent = pq.remove();
                c = nextEvent->getCustomer();
                if( c->getStatus() == -1 ){  // arrival event
```

vu Elon musk files

```
                int arrTime = nextEvent->getEventTime();
                int duration = c->getTransactionDuration();
                int customerNo = c->getCustomerNumber();
                processArrival(data, customerNo,
                        arrTime, duration , nextEvent);
        }
        else {  // departure event
                int qindex = c->getStatus();
                int departTime = nextEvent->getEventTime();
                processDeparture(qindex, departTime, nextEvent);
        }
    }
```

Let's discuss the function readNewCustomer (). This function is used to read the data from the file.

```
void readNewCustomer(ifstream& data)
{
        int hour,min,duration;
        if (data >> hour >> min >> duration) {
                customerNo++;
                Customer* c = new Customer(customerNo,
                        hour*60+min, duration);
                c->setStatus( -1 ); // new arrival
                Event* e = new Event(c, hour*60+min );
```

vu Elon musk files

```
            pq.insert( e ); // insert the arrival event
        }
        else {
            data.close(); // close customer file
        }
}
```

Let's see the function processArrival(). We have decided that when the customer arrives and no teller is available, he will go to the shortest queue.

```
int processArrival(ifstream &data, int customerNo,
                   int arrTime, int duration,
                   Event* event)
{
    int i, small, j = 0;
    // find smallest teller queue
  small = q[0].length();
  for(i=1; i < 4; i++ )
            if( q[i].length() < small ){
     small = q[i].length(); j = i;
            }
    // put arriving customer in smallest queue
    Customer* c = new Customer(customerNo, arrTime,
                               duration );
    c->setStatus(j);  // remember which queue the customer goes in
```

vu Elon musk files

```
        q[j].enqueue(c);
        // check if this is the only customer in the.
        //  queue. If so, the customer must be marked for
        // departure by placing him on the event queue.
        if( q[j].length() == 1 ) {
                c->setDepartureTime( arrTime+duration);
                Event* e = new Event(c, arrTime+duration );
                pq.insert(e);
        }
        // get another customer from the input
        readNewCustomer(data);
}
```

There may be a case that before leaving the bank, more persons arrive and they have to wait in the queue for their turn. We handle this scenario in the departure routine. The code is:

```
int processDeparture(     int qindex, int departTime,
                          Event* event)
{
        Customer* cinq = q[qindex].dequeue();
        int waitTime = departTime - cinq->getArrivalTime();
        totalTime = totalTime + waitTime;
        count = count + 1;
        // if there are any more customers on the queue, mark the
```

vu Elon musk files

```
// next customer at the head of the queue for departure
// and place him on the eventList.
if( q[qindex].length() > 0 ) {
        cinq = q[qindex].front();
        int etime = departTime + cinq->getTransactionDuration();
        Event* e = new Event( cinq, etime);
        pq.insert( e );
}}
// print the final avaerage wait time.
double avgWait = (totalTime*1.0)/count;
cout << "Total time:    " << totalTime << endl;
cout << "Customer:      " << count << endl;
cout  << "Average wait: " << avgWait << endl;
```

Data Structures Lecture No. 11

Implementation of Priority Queue

In the priority queue, we put the elements in the queue to get them from the queue with a priority of the elements. Following is the C++ code of the priority queue.

Priority Queue

```
#include "Event.cpp"
#define PQMAX 30
classPriorityQueue {
public:
```

vu Elon musk files

```
        PriorityQueue() {
                size = 0; rear = -1;
        };
        ~PriorityQueue() {};

        int full(void)
        {
return ( size == PQMAX ) ? 1 : 0;
        };

        Event* remove()
        {
                if( size > 0 ) {
          Event* e = nodes[0];
                        for(int j=0; j < size-2; j++ )
                                nodes[j] = nodes[j+1];

                        size = size-1; rear=rear-1;
                        if( size == 0 ) rear = -1;

                        return e;
                }
                return (Event*)NULL;
```

vu Elon musk files

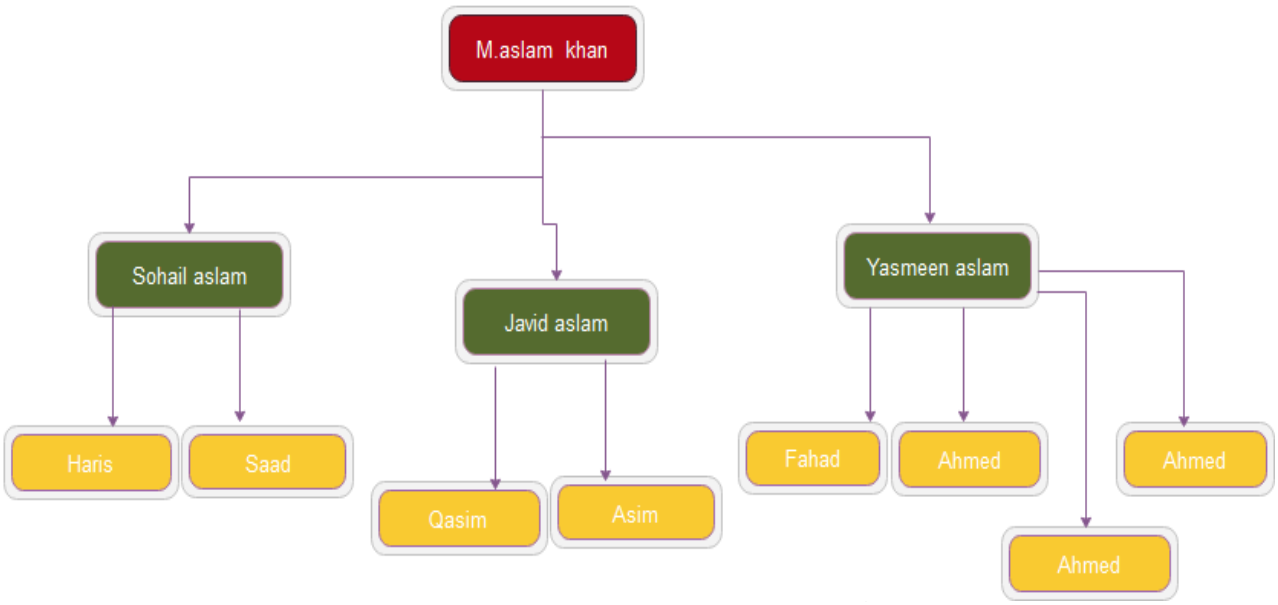
```
            cout<< "remove - queue is empty." <<endl;
        };
int insert(Event* e)
        {
            if( !full() ) {
                rear = rear+1;
                nodes[rear] = e;
                size = size + 1;
                sortElements(); // in ascending order
                return 1;
            }
            cout<< "insert queue is full." <<endl;
            return 0;
        };

        int length() { return size; };
};
```

vu Elon musk files

Tree

Tree is an important data structure .In this structures the elements are in a line. We put and get elements in and from a stack in linear order. Queue is also a linear data structure as a line is developed in it. Tree is one of the non-linear data structures. The figure given below is showing a genealogy tree of a family.

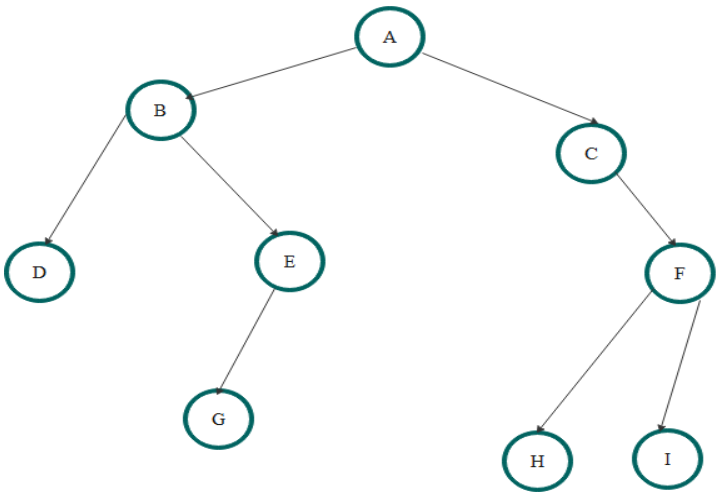


In this genealogy tree, the node at the top of the tree is *Muhammad Aslam Khan* i.e. the head of the family. There are three nodes under this one. These are *Sohail Aslam*,
Javed Aslam and *Yasmeen Aslam*. Then there are nodes under these three nodes i.e.
the sons of these three family members.

Binary Tree

"A binary tree is a finite set of elements that is either empty or is partitioned into three
disjoint subsets."The first subset contains a single element called the root of the tree. The other two subsets are themselves binary trees called the left and right sub-trees". Each element of a binary tree is called a node of the tree.

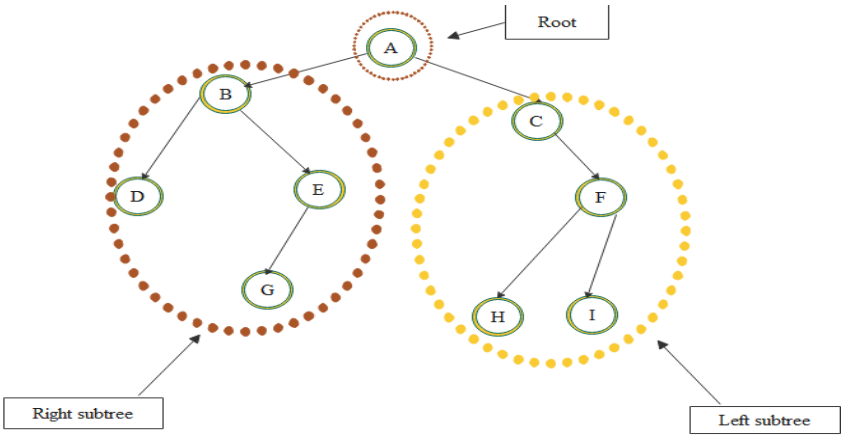
vu elon musk



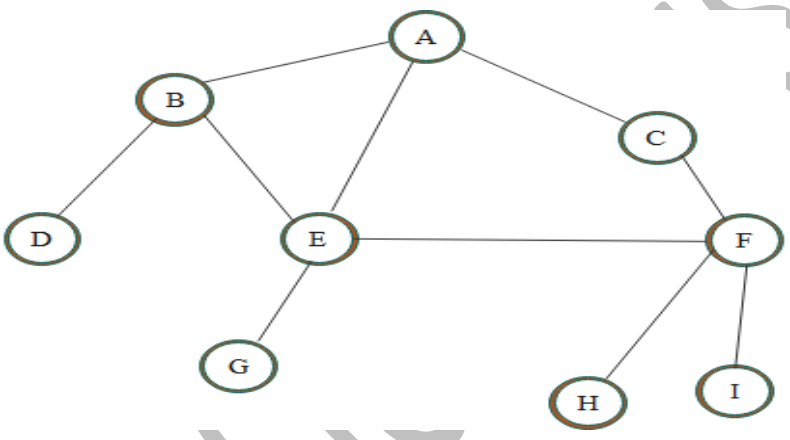
There are nine nodes are A, B,C, D, E, F, G, H, and I. The node A is at the top of the tree. There are two lines from the node A to left and right sides towards node B and C.There are also two lines from node B to left and right, leading to the nodes D and E. node, E there is only one line that is to the left of the node and leads to node G. node C towards the node F on right side of the node C. Then there are two lines from node F that leads to the nodes H and I on left and right side.
Now we analyze this tree according to the mathematical definition of the binary tree. The node A is the root of the tree. And tree structure (i.e. the nodes B, D, E, G and their relation) on the left side of the node A is a sub-tree, called the *Left subtree*.

Similarly the nodes (C, F, H and I) on the right side of the node A comprise the subtree
termed as *Right subtree*. Thus we made three parts of the tree. One part containsonly one node i.e. A while the second part has all the nodes on left side of A, called as *Left subtree*. Similarly the third part is the *Right subtree*, containing the nodes on right side of A.

www.elonmusk.files

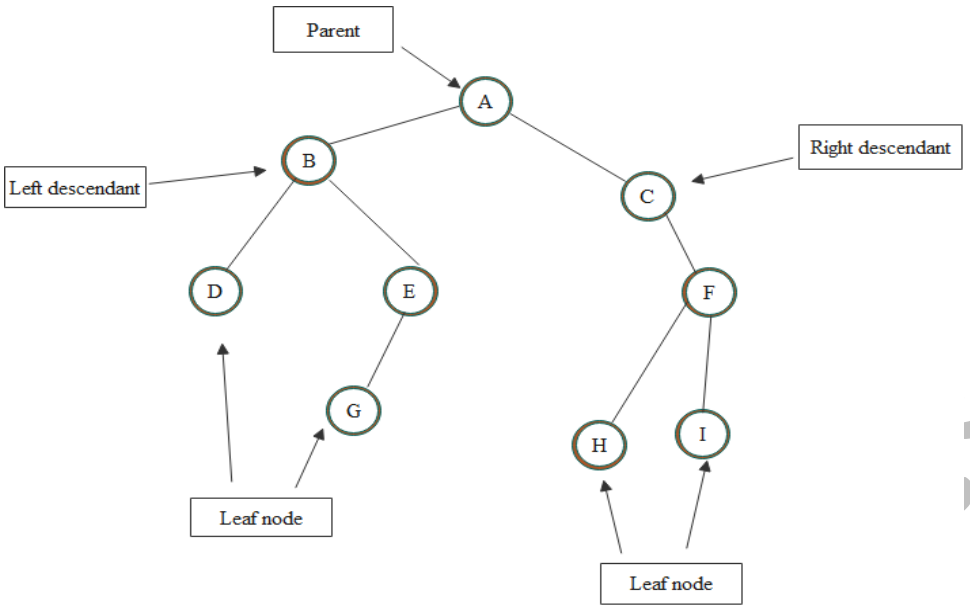


Similarly if we put other links between different nodes, then the structure thus
developed will not be a tree. The following figure is also an example of a structure that is not a tree as there are multiple links between the different nodes.



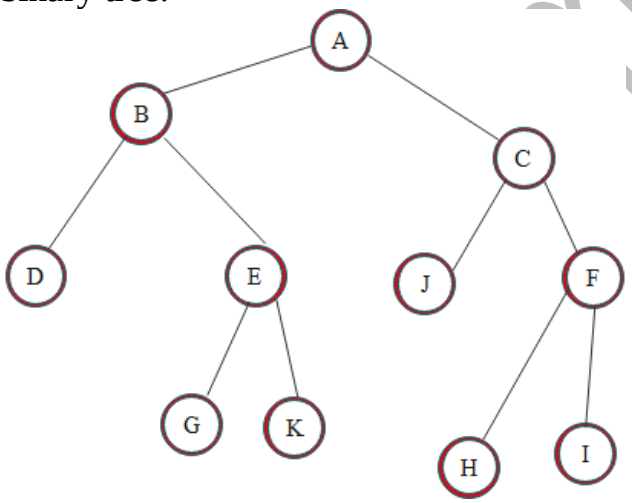
Terminologies of a binary tree

In the relationship of these nodes, A is the parent node with B and C as the left and right descendants respectively. C is the right descendant of A. we look at the node B where the node D is its left descendant and E is its right descendant. We can use the words descendant and child interchangeably. Now look at the nodes D, G, H and I. These nodes are said leaf nodes as there is no descendant of these nodes. We are just introducing the terminology here. In the algorithms, we use the words root or parent, child or descendant.



Strictly Binary Tree

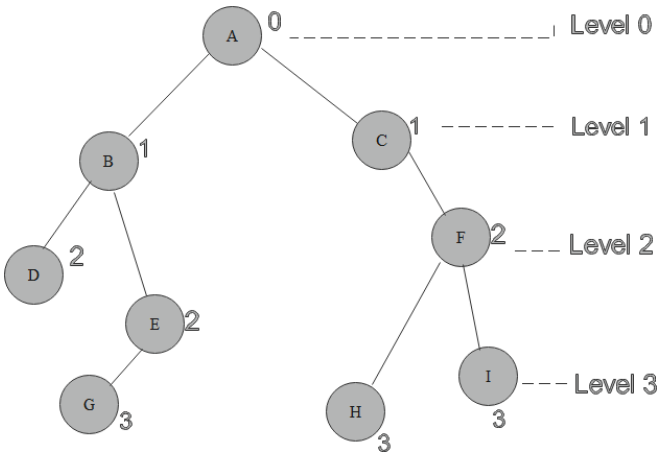
Binary tree is said to be a strictly binary tree if every non-leaf node in a binary tree has non-empty left and right subtrees. Now all the non-leaf nodes (A, B, C, E and F) have their left and right children so according to the definition, it is a strictly binary tree.



Level

The level of a node in a binary tree is defined as follows:

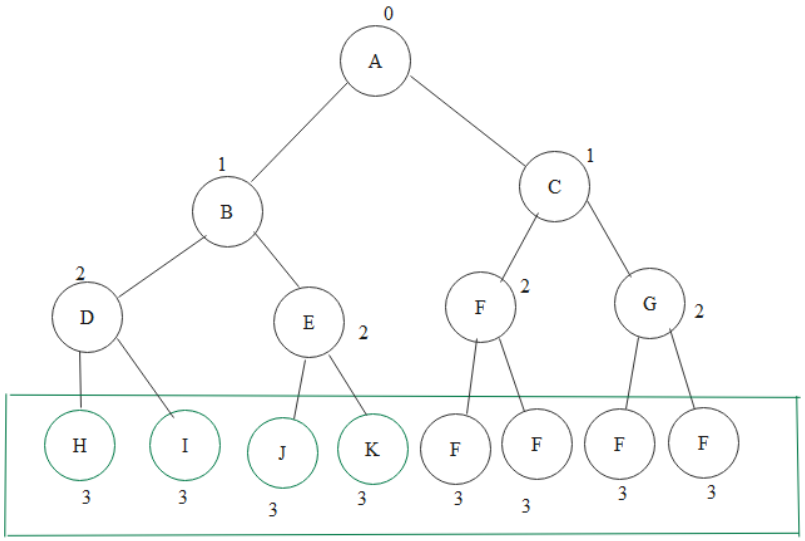
- Root has level 0,
- Level of any other node is one more than the level its parent (father).
- The *depth* of a binary tree is the maximum level of any leaf in the tree.



In the above figure, we have mentioned the level of each node. Thus we see that tree has multi levels in the downward direction. The levels of the tree increase, as the tree grows downward more. The number of nodes also increases. By seeing the level of a tree, we can tell the depth of the tree. If we put level with each node of the binary tree, the depth of a binary tree is the maximum level. In the figure, the deepest node is at level 3 the maximum level is 3. So the depth of this binary tree is 3.

Complete Binary Tree

If there are left and right subtrees for each node in the binary tree. it becomes a complete binary tree.

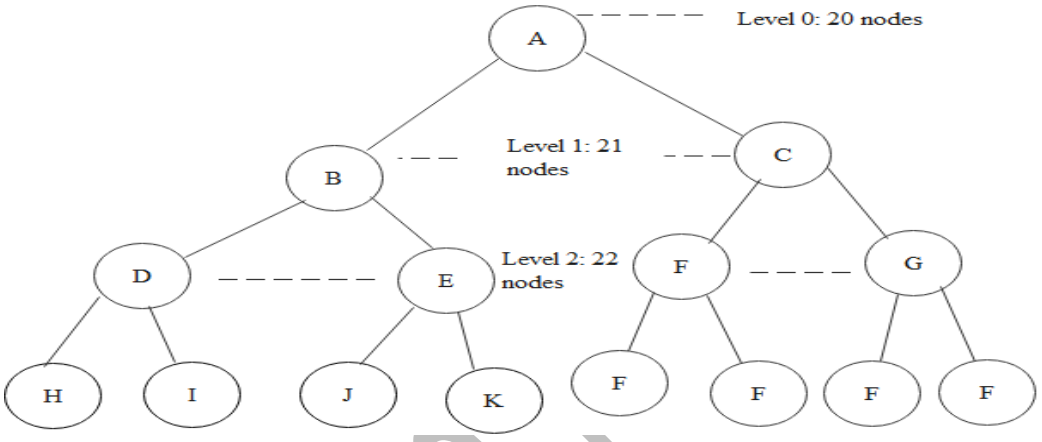


Now look at the tree, the leaf nodes of the tree are at level 3 and are H, I, J, K, L, M,

musk files

N and O. There is no such a leaf node that is at some level other than the depth level *d*
i.e. 3. All the leaf nodes of this tree are at level 3. So this is a complete binary tree. The property of the binary tree is that its node can have a maximum of two subtrees, called as left and right subtrees. The nodes increase at each level, as the tree grows downward.

In a complete binary tree, the nodes increase in a particular order. If we reconsider the previous tree. we note that the number of node at level 0 is 1. We can say it 20, as 20 is equal to 1. Down to this, the node is the level 1 where the number of nodes is 2, which we can say 21, equal to 2. Now at the next level (i.e. level 2), the number of nodes is 4 that mean 22. Finally the number of nodes at level 3 is 8, which can be called as 23.



By observing the number of nodes at a particular level, we come to the conclusion That the number of nodes at a level is equal to the level number raising to the power of
two.

- Applying this formula to each level while going to the depth of the tree.And we can calculate the total number of nodes in a complete binary tree of depth *d* by adding the number of nodes at each level.

$$2^0 + 2^1 + 2^2 + \ldots\ldots + 2^d = \Sigma^d_{j=0}\, 2^j = 2^{d+1} - 1$$

- if there is a complete binary tree of depth 4, thetotal number of nodes in it will be calculated by putting the value of *d* equal to 4. It will be calculated as under.

$$2^{4+1} - 1 = 25 - 1 = 32 - 1 = 31$$

Thus the total number of nodes in the complete binary tree of depth 4 is 31.

Level of a Complete Binary Tree

We can find the depth of a complete binary tree if we know the total number of nodes. If we have a complete binary tree with *n* total nodes, then by the equation of the total number of nodes.

Total number of nodes = $2^{d+1} - 1 = n$
To find the value of *d*, we solve the above equation as under
$2^{d+1} - 1 = n$
$2^{d+1} = n + 1$
$d + 1 = \log_2 (n + 1)$
$d = \log_2 (n + 1) - 1$

Operations on Binary Tree

We can define different operations on binary trees.

1. left(*p*) returns pointer to the left subtree
2. right(*p*) returns pointer to right subtree
3. parent(*p*) returns the father of *p*
4. brother(*p*) returns brother of *p*.
5. info(p) returns content of the node.

vu elon musk files

Lecture No. 12

Operations on Binary Tree

There are a number of operations that can be defined for a binary tree.

If *p* is pointing to a node in an existing tree then

1. left(*p*) returns pointer to the left subtree
2. right(*p*) returns pointer to right subtree
3. parent(*p*) returns the father of *p*
4. brother(*p*) returns brother of *p*.

5. Info(p) returns content of the node.
6. In order to construct a binary tree, the following can be useful:
7. setLeft(p,x) creates the left child node of p. The child node contains the info 'x'.
8. setRight(p,x) creates the right child node of p. The child node contains the info 'x'.

Applications of Binary Tree

One way of finding duplicates is to compare each number with all those that precede it.

- 14, 15, 4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9, 14, 5

- 14, 15, 4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9, 14, 5



Searching for Duplicates

- If the list of numbers is large and is growing, this procedure involves a large number of comparisons.
- A linked list could handle the growth but the comparisons would still be large.
- The number of comparisons can be drastically reduced by using a binary tree.
- The tree grows dynamically like the linked list.
- The binary tree is built in a special way.
- The first number in the list is placed in a node that is designated as the root of a binary tree.
- Initially, both left and right subtrees of the root are empty.

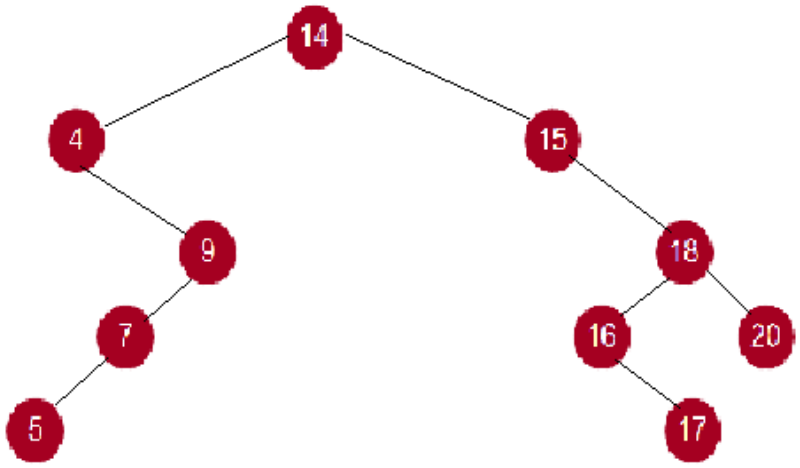
www.elonmusk files

- We take the next number and compare it with the number placed in the root.

If it is the same then we have a duplicate.

- Otherwise, we create a new tree node and put the new number in it.
- The new node is made the left child of the root node if the second number is less than the one in the root.

The new node is made the right child if the number is greater than the one in the root.



9, 14, 5

C++ Implementation of Binary Tree

The code given below is for the file treenode.cpp.

```
#include <stdlib.h>
template <class Object>
class TreeNode {
public:
    // constructors
    TreeNode()
```

```
        {
                this->object = NULL;
                this->left = this->right = NULL;
        };
      TreeNode( Object* object )
        {
                this->object = object;
                this->left = this->right = NULL;
        };
Object* getInfo()
{
    return this->object;
};
void setInfo(Object* object)
{
    this->object = object;
};
TreeNode* getLeft()
{
    return left;
};
void setLeft(TreeNode *left)
{
```

vu Elon musk files

```
    this->left = left;
};
TreeNode *getRight()
{
    return right;
};
void setRight(TreeNode *right)
{
    this->right = right;
};

int isLeaf( )
{
    if( this->left == NULL && this->right == NULL )
            return 1;
    return 0;
};
private:
    Object*   object;
    TreeNode* left;
    TreeNode* right;
}; // end class TreeNode
```

The code of main program file containing the main() and insert() functions.

vu Elon musk files

```
#include <iostream>
#include <stdlib.h>
#include "TreeNode.cpp"
int main(int argc, char *argv[])
{
    int x[] = { 14, 15, 4, 9, 7, 18, 3, 5, 16,4, 20, 17,
                    9, 14,5, -1};
    TreeNode<int>* root = new TreeNode<int>();
    root->setInfo( &x[0] );
    for(int i=1; x[i] > 0; i++ )
    {
            insert(root, &x[i] );
    }
}
void insert(TreeNode<int>* root, int* info)
{
   TreeNode<int>* node = new TreeNode<int>(info);
   TreeNode<int> *p, *q;
   p = q = root;
   while( *info != *(p->getInfo()) && q != NULL )
   {
      p = q;
      if( *info < *(p->getInfo()) )
```

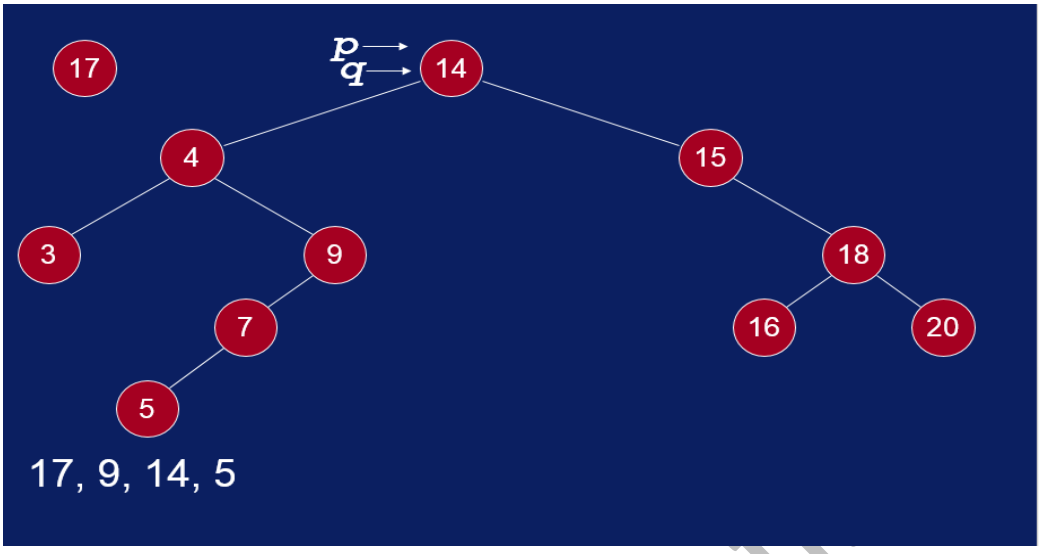
vu Elon musk files

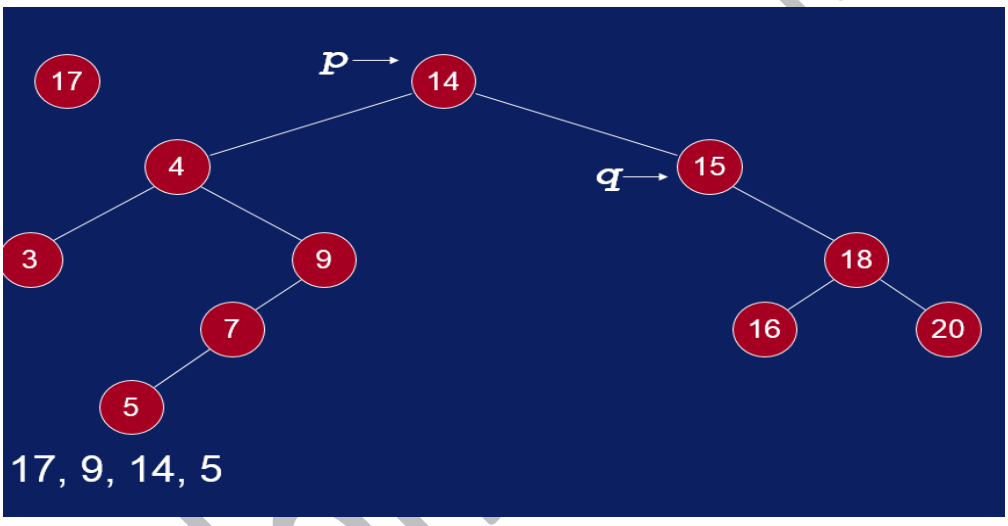
```
            q = p->getLeft();
        else
            q = p->getRight();
    }
if( *info == *(p->getInfo()) ){
        cout << "attempt to insert duplicate: "
                    << *info << endl;
        delete node;
    }
    else if( *info < *(p->getInfo()) )
        p->setLeft( node );
    else
        p->setRight( node );
} // end of insert
```

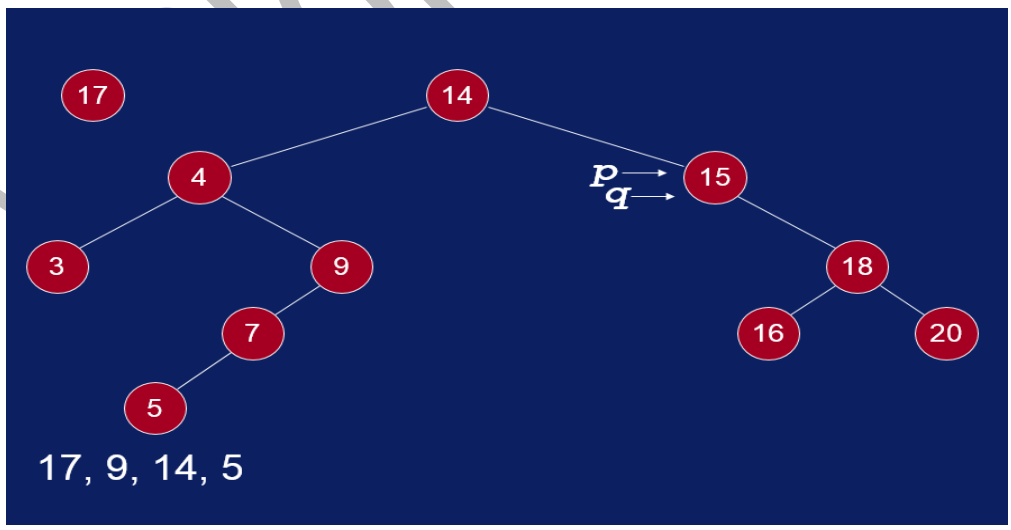
Trace of insert

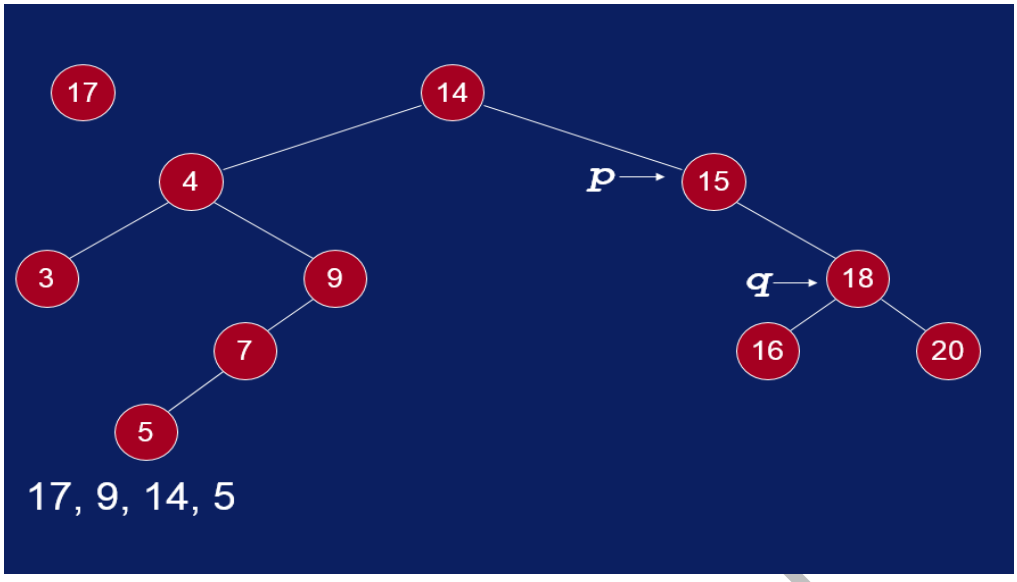
To execute this algorithm manually to insert the remaining numbers in the tree. This will make the understanding more clear. This tree algorithm is going to be used rigorously in the future and complete understanding of it will make the complex future implementations easier to comprehend.

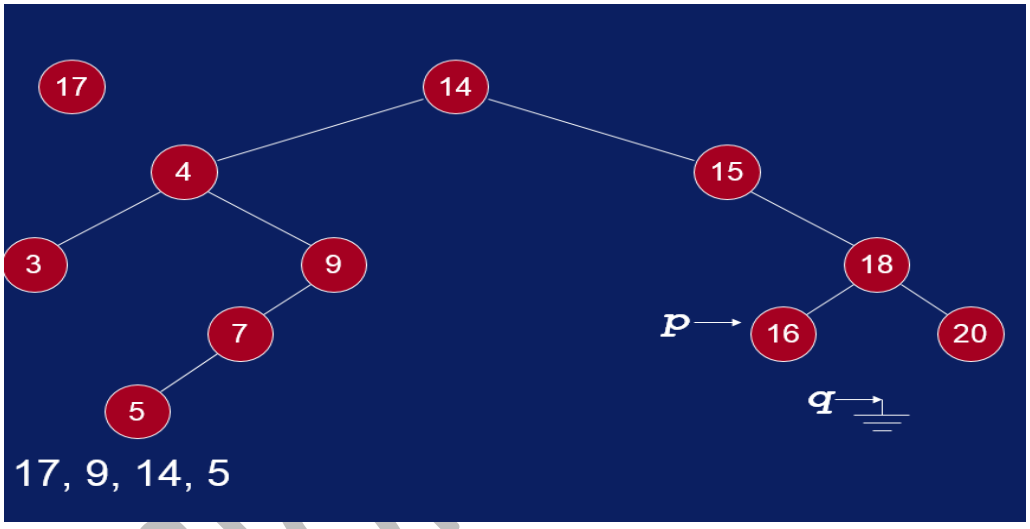
vu Elon musk files

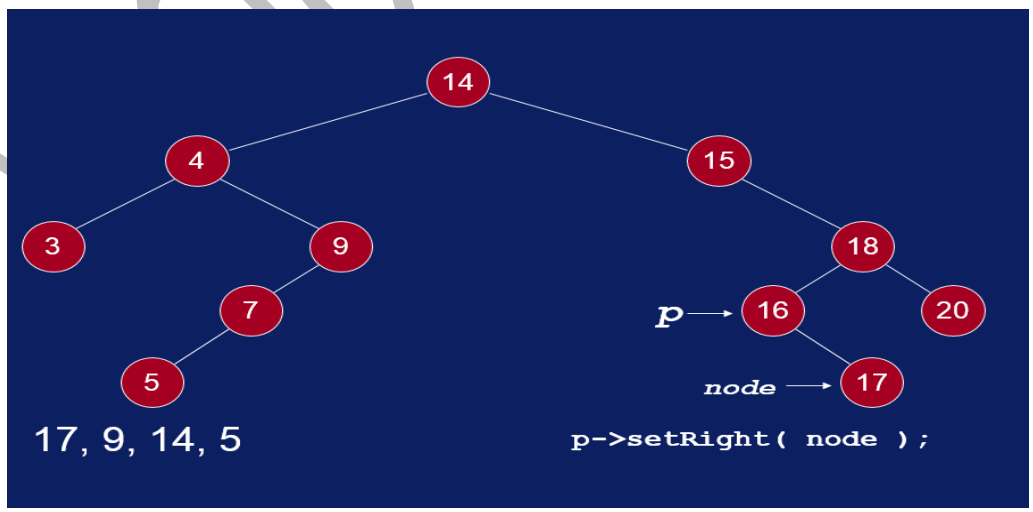

p
q
17
14
4
15
3
9
18
7
16
20
5
17, 9, 14, 5


p
17
14
4
q
15
3
9
18
7
16
20
5
17, 9, 14, 5


17
14
4
p
q
15
3
9
18
7
16
20
5
17, 9, 14, 5


17
14
4
p
15
3
9
q
18
7
16
20
5
17, 9, 14, 5


17
14
4
15
3
9
18
p
7
16
20
q
5
17, 9, 14, 5


14
4
15
3
9
18
p
7
16
20
node
5
17
17, 9, 14, 5
p->setRight(node);

Lecture No. 13

Cost of Search

- Given that a binary tree is level *d* deep. How long does it take to find out whether a number is already present?
- Consider the insert(17) in the example tree.
- Each time around the while loop, we did one comparison.
- After the comparison, we moved a level *down*.
- With the binary tree in place, we can write a routine *find(x)* that returns true if the number *x* is present in the tree, false otherwise.
- How many comparison are needed to find out if *x* is present in the tree?
- We do one comparison at each level of the tree until either *x* is found or q becomes NULL.
- If the binary tree is built out of *n* numbers, how many comparisons are needed to find out if a number *x* is in the tree?
- Recall that the depth of the complete binary tree built using '*n*' nodes will be $log_2(n+1) - 1$.
- For example, for n=100,000, $\log_2(100001)$ is less than 20; the tree would be 20 levels deepIf the tree is complete binary or nearly complete, searching through 100,000 numbers will require a maximum of 20 comparisons.
- Or in general, approximately $log_2(n)$.
- Compare this with a linked list of 100,000 numbers.

Cost of Search

- If the tree is complete binary or nearly complete, searching through 100,000 numbers will require a maximum of 20 comparisons.

vu Elon musk files

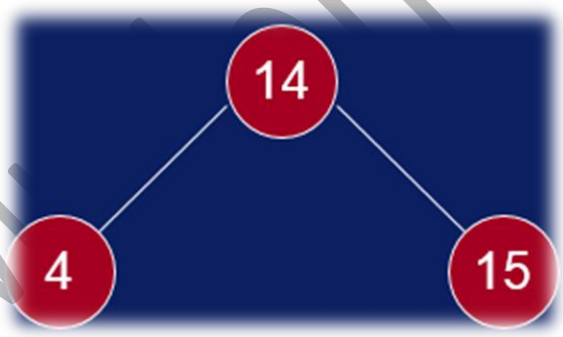
- Or in general, approximately $log_2(n)$.
- Compare this with a linked list of 100,000 numbers. The comparisons required could be a maximum of *n*.
- A binary tree with the property that items in the left subtree are smaller than the root and items are larger or equal in the right subtree is called a *binary search tree* (BST).
- The tree we built for searching for duplicate numbers was a binary search tree.

  BST and its variations play an important role in searching algorithms

  Traversing a Binary Tree
- Suppose we have a binary tree, ordered (BST) or unordered.
- We want to print all the values stored in the nodes of the tree.

  In what order should we print them?

  These are the Ways to print a 3 node tree:



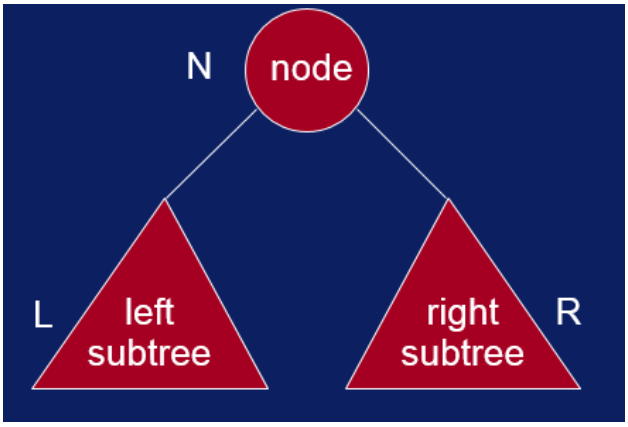
(4, 14, 15), (4,15,14)

(14,4,15), (14,15,4)

(15,4,14), (15,14,4)

vision musk files

In case of the general binary tree



(L,N,R), (L,R,N)

(N,L,R), (N,R,L)

(R,L,N), (R,N,L)

Three common ways are:

Preorder: (N,L,R)

Inorder: (L,N,R)

Postorder: (L,R,N)

Traversing a Binary Tree

```
void preorder(TreeNode<int>* treeNode)
{
  if( treeNode != NULL )
  {
     cout << *(treeNode->getInfo())<<" ";
     preorder(treeNode->getLeft());
     preorder(treeNode->getRight());
  }
```

vu Elon musk files

```
}
void inorder(TreeNode<int>* treeNode){
   if( treeNode != NULL )
   {
      inorder(treeNode->getLeft());
      cout << *(treeNode->getInfo())<<" ";
      inorder(treeNode->getRight());
   }
}
cout << "inorder: ";   preorder( root);
cout << "inorder: ";   inorder( root );
cout << "postorder: "; postorder( root );
```

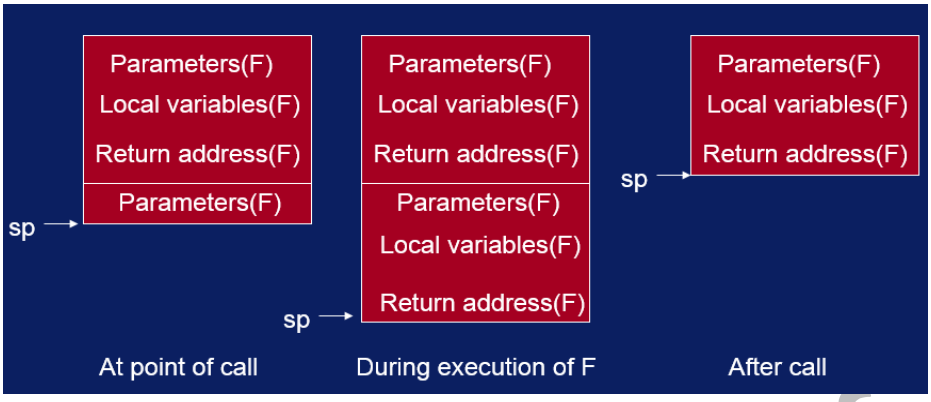
Recursive Call

- **Recall that a stack is used during function calls.**
- **The caller function places the arguments on the stack and passes control to the called function.**
- **Local variables are allocated storage on the call stack.**
- **Calling a function itself makes no difference as far as the call stack is concerned.**

Stack Layout during a call

- **Here is the stack layout when function F calls function F (recursively)**

vu Elon musk files



Lecture No. 14

Recursion: preorder

preorder(14)

14

..preorder(4)

4

....preorder(3)

3

.....preorder(null)

.....preorder(null)

....preorder(9)

9

.....preorder(7)

7

.....preorder(5)

vu elon musk files

5

.....preorder(null)

.....preorder(null)

.....preorder(null)

.....preorder(null

..preorder(15)

15

....preorder(null)

....preorder(18)

18

.....preorder(16)

16

.....preorder(null)

.....preorder(17)

17

.....preorder(null)

.....preorder(null)

.....preorder(20)

20

.....preorder(null)

.....preorder(null)

inorder(14)

..inorder(4)

vu Elon musk files

....inorder(3)

......inorder(null)

3

......inorder(null)

4

....inorder(9)

......inorder(7)

........inorder(5)

..........inorder(null)

5

..........inorder(null)

7

........inorder(null)

9

......inorder(null)

14

..inorder(15)

....inorder(null)

15

....inorder(18)

......inorder(16)

........inorder(null)

vu £lon musk files

16

.....inorder(17)

.....inorder(null)

17

.....inorder(null)

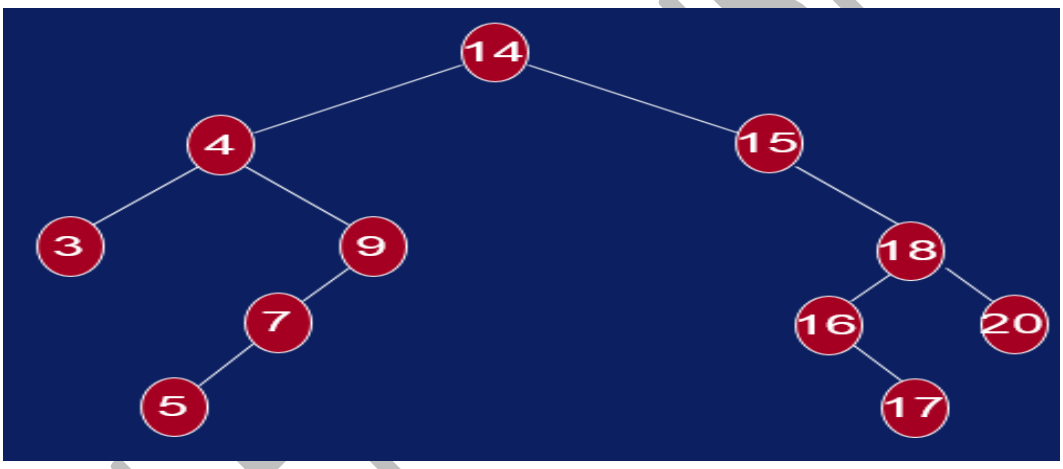
18

.....inorder(20)

.....inorder(null)

20

.....inorder(null)



Recursion: inorder

inorder(14)

..inorder(4)

....inorder(3)

.....inorder(null)

3

......inorder(null)

4

....inorder(9)

......inorder(7)

........inorder(5)

..........inorder(null)

5

..........inorder(null)

7

........inorder(null)

9

......inorder(null)

14

..inorder(15)

....inorder(null)

15

....inorder(18)

......inorder(16)

........inorder(null)

16

........inorder(17)

..........inorder(null)

vu Elon musk files

17

.....inorder(null)

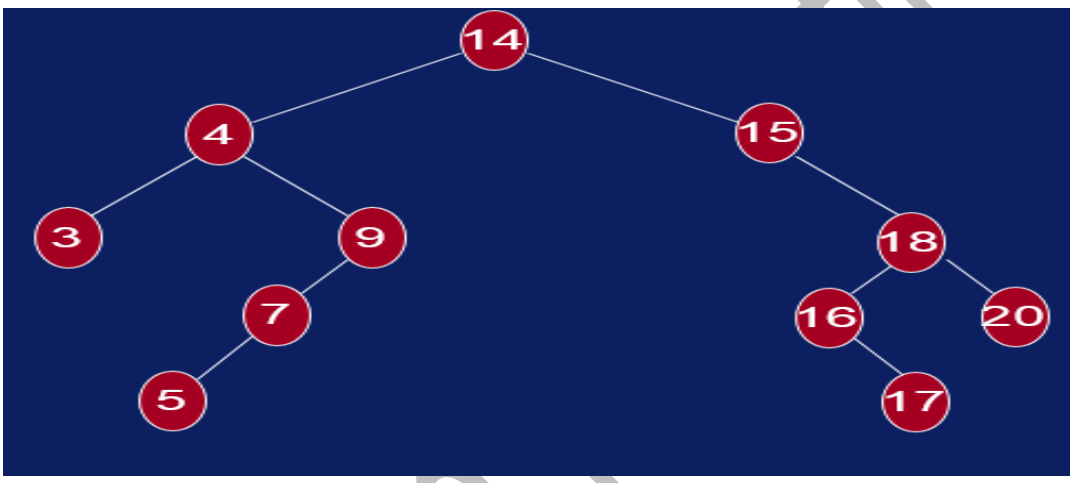
18

.....inorder(20)

.....inorder(null)

20

.....inorder(null)



Non Recursive Traversal

- We can implement non-recursive versions of the preorder, inorder and postorder traversal by using an explicit stack.
- The stack will be used to store the tree nodes in the appropriate order.
- Here, for example, is the routine for inorder traversal that uses a stack.

```
void inorder(TreeNode<int>* root)
    {
   Stack<TreeNode<int>* > stack;
```

```
    TreeNode<int>* p;
    p = root;
    do
    {
        while( p != NULL )
        {
            stack.push( p );
            p = p->getLeft();
        }
        // at this point, left tree is empty
if( !stack.empty() )
                {
        p = stack.pop();
        cout << *(p->getInfo()) << " ";
                    // go back & traverse right subtree
        p = p->getRight();
                }
  } while ( !stack.empty() || p != NULL );
}
```

Traversal Trace

Recursive inorder	Non-recursive inorder

vu Elon musk files

inorder(14)	**push (14)**
..inorder(4)	**.. push (4)**
....inorder(3)	**.... push (3)**
3	**3**
4	**4**
..inorder(9)	**.. push (9)**
....inorder(7)	**.... push (7)**
......inorder(5)	**...... push (5)**
5	**5**
7	**7**
9	**9**
14	**14**
inorder(15)	**push (15)**
15	**15**
inorder(18)	**push (18)**
..inorder(16)	**.. push (16)**
16	**16**
..inorder(17)	**.. push (17)**
17	**17**
18	**18**
inorder(20)	**push (20)**
20	**20**

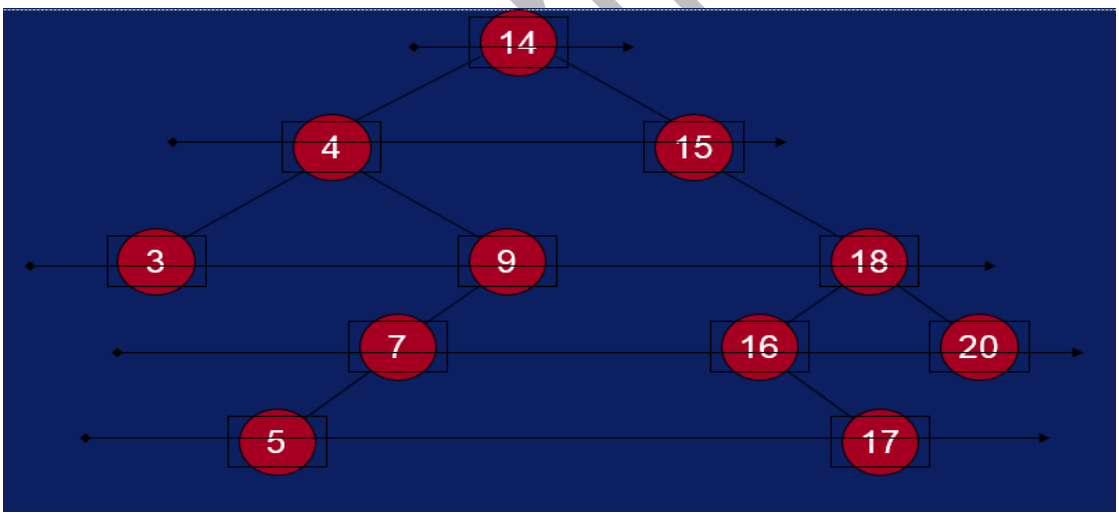
vu Elon musk files

Level-order Traversal

- There is yet another way of traversing a binary tree that is not related to recursive traversal procedures discussed previously.
- In level-order traversal, we visit the nodes at each level before proceeding to the next level.

At each level, we visit the nodes in a left-to-right order:

Level-order: 14 4 15 3 9 18 7 16 20 5 17

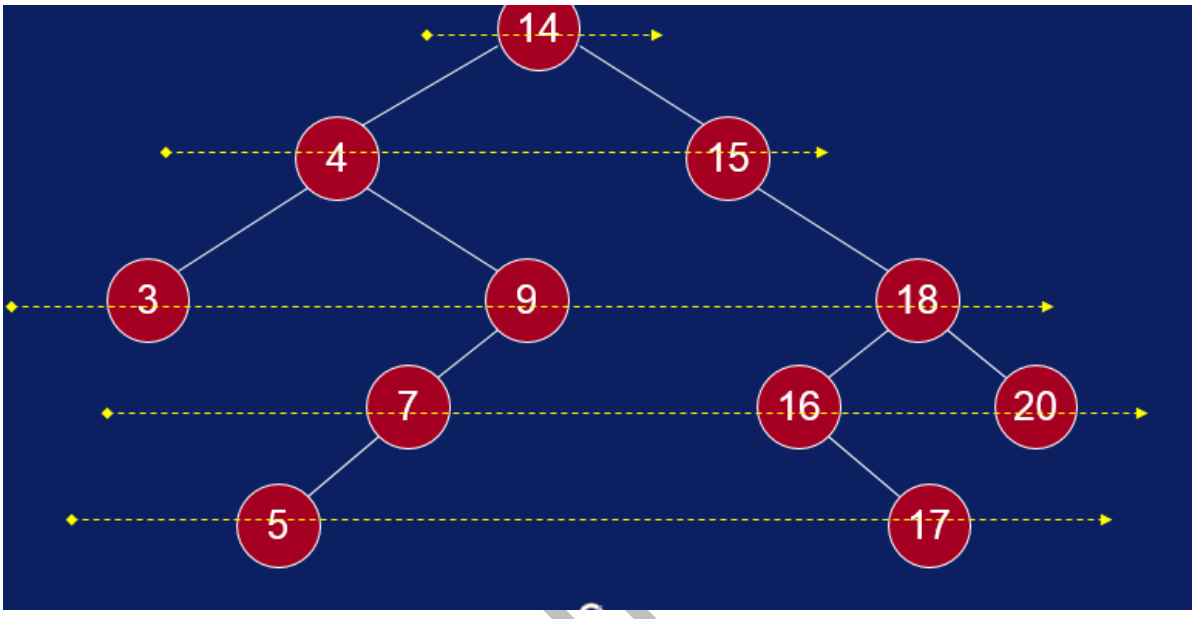


musk files

Lecture No. 15

Level-order Traversal of a Binary Tree

- There is yet another way of traversing a binary tree that is not related to recursive traversal procedures discussed previously.
- In level-order traversal, we visit the nodes at each level before proceeding to the next level.
- At each level, we visit the nodes in a left-to-right order.



Level-order: 14 4 15 3 9 18 7 16 20 5 17

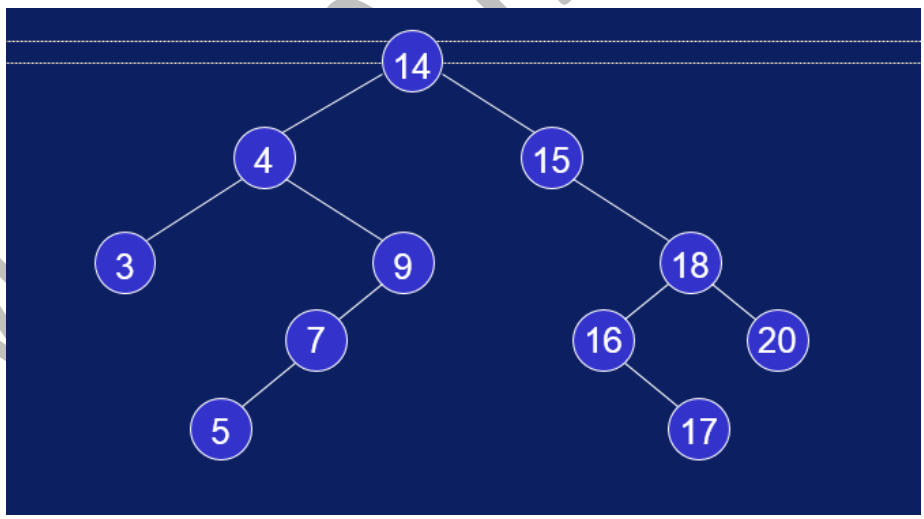
How do we do level-order traversal?

Surprisingly, if we use a queue instead of a stack, we can visit the nodes in level-order.

- Here is the code for level-order traversal

```
void levelorder(TreeNode<int>* treeNode)
{
   Queue<TreeNode<int>* > q;
```

```
    if( treeNode == NULL ) return;
    q.enqueue( treeNode);
    while( !q.empty() )
    {
       treeNode = q.dequeue();
       cout << *(treeNode->getInfo()) << " ";
       if(treeNode->getLeft() != NULL )
                    q.enqueue( treeNode->getLeft());
       if(treeNode->getRight() != NULL )
                    q.enqueue( treeNode->getRight());
    }
    cout << endl;
}
```



Output: 14 4 15 3 9 18 7 16 20 5 17.

Storing Other Types of Data in Binary Tree

www.musk files

- The examples of binary trees so far have been storing integer data in the tree node.
- This is surely not a requirement. Any type of data can be stored in a tree node.

  Here, for example, is the C++ code to build a tree with character strings.

  Binary Search Tree with Strings

```
void wordTree()
{
        TreeNode<char>* root = new TreeNode<char>();
        static char* word[] = "babble", "fable", "jacket",
 "backup", "eagle","daily","gain","bandit","abandon",
         "abash","accuse","economy","adhere","advise","cease",
         "debunk","feeder","genius","fetch","chain", NULL};
        root->setInfo( word[0] );

        for(i=1; word[i]; i++ )
              insert(root, word[i] );
        inorder( root ); cout << endl;
}
void insert(TreeNode<char>* root, char* info)
{
   TreeNode<char>* node = new TreeNode<char>(info);
   TreeNode<char> *p, *q;
   p = q = root;
```

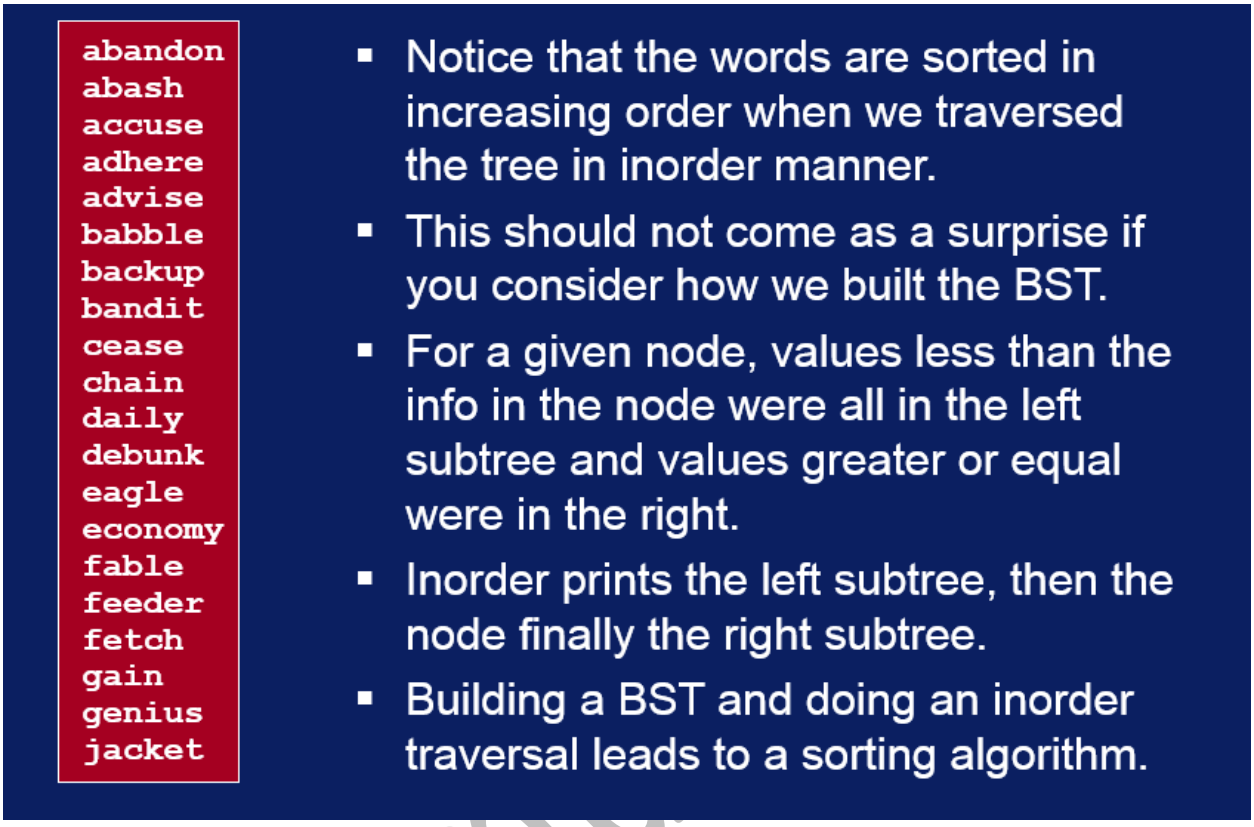
vu Elon musk files

```
    while( strcmp(info, p->getInfo()) != 0 && q != NULL )
    {
       p = q;
       if( strcmp(info, p->getInfo()) < 0 )
          q = p->getLeft();
       else
          q = p->getRight();
    }
if( strcmp(info, p->getInfo()) == 0 ){
    cout << "attempt to insert duplicate: " << *info
       << endl;
    delete node;
       }
       else if( strcmp(info, p->getInfo()) < 0 )
    p->setLeft( node );
       else
    p->setRight( node );
```

vu Elon musk files

}

Output



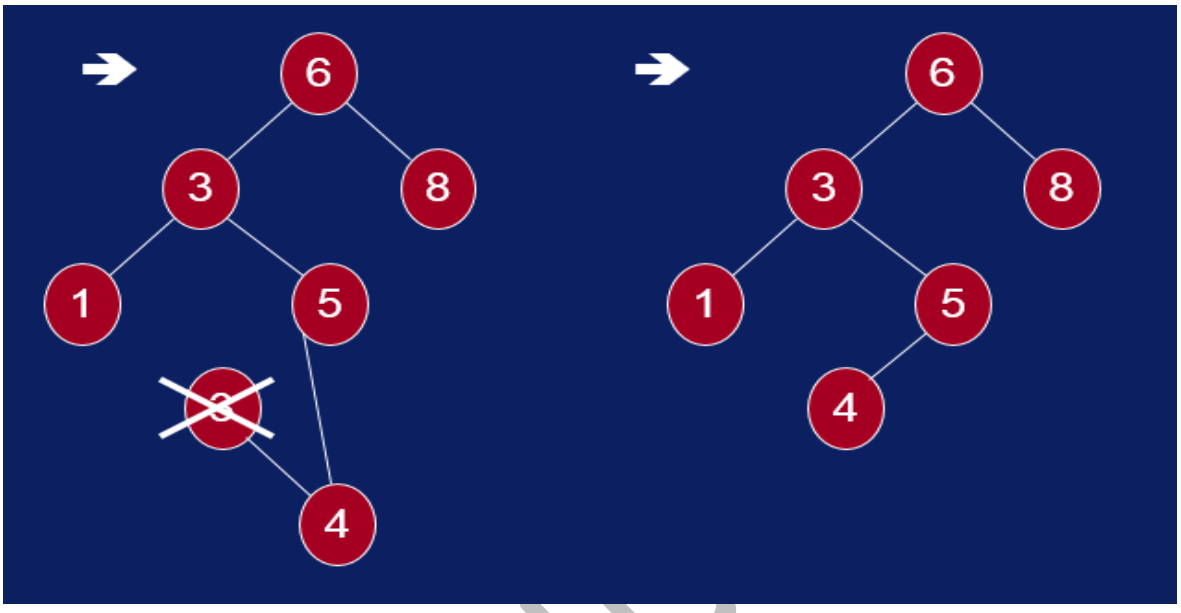
Deleting a Node From BST

- As is common with many data structures, the hardest operation is deletion.
- Once we have found the node to be deleted, we need to consider several possibilities.
- If the node is a *leaf*, it can be deleted immediately.
- If the node has one child, the node can be deleted after its parent adjusts a pointer to bypass the node and connect to inorder successor.
- The inorder traversal order has to be maintained after the delete.
- The complicated case is when the node to be deleted has both left and right subtrees.
- The strategy is to replace the data of this node with the smallest data of the right subtree and recursively delete that node.

Deleting a node in BST

- locate inorder successor.
- Inorder successor will be the left-most node in the right subtree of 2.

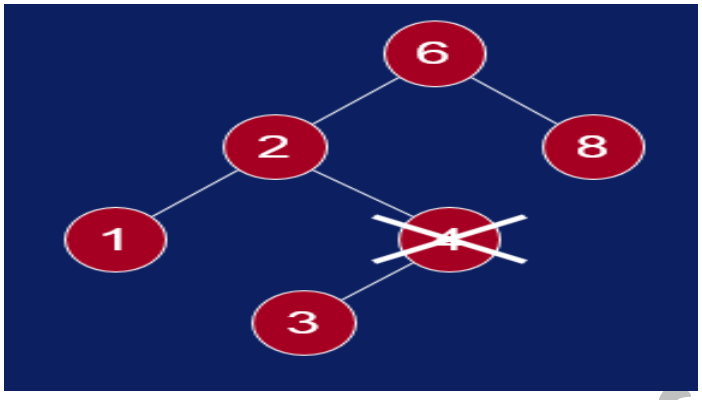
- The inorder successor will not have a left child because if it did, that child would be the left-most node.
- copy data from inorder successor.
- remove the inorder successor



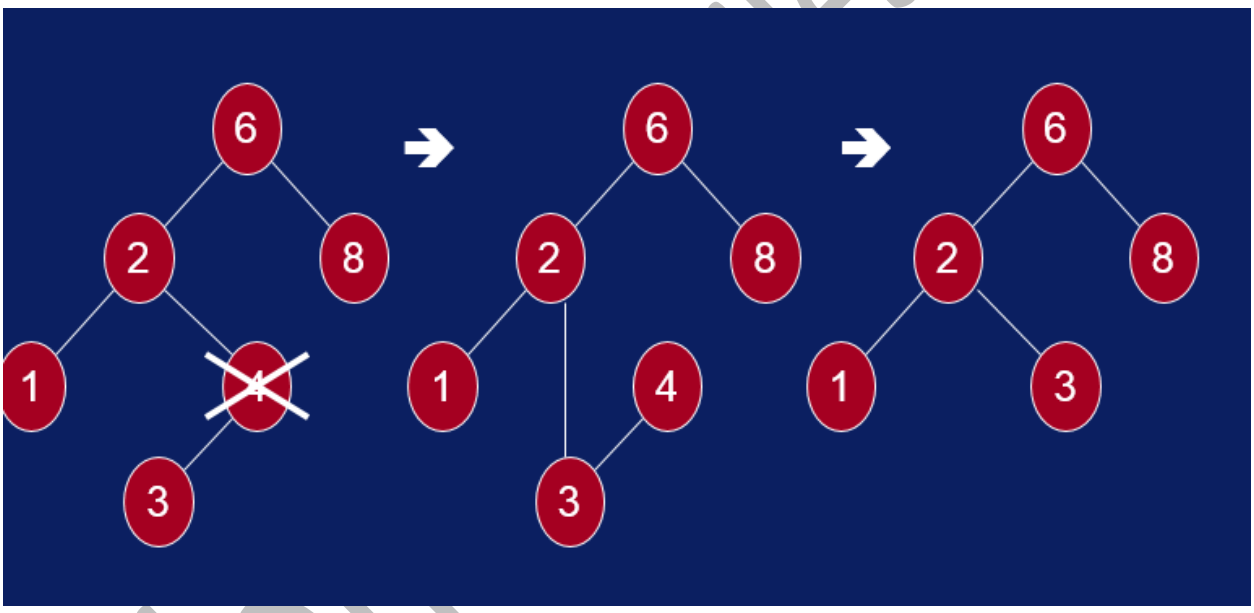
Lecture No. 16

Deleting a node in BST

- If the node has one child, the node can be deleted after its parent adjusts a pointer to bypass the node and connect to inorder successor.



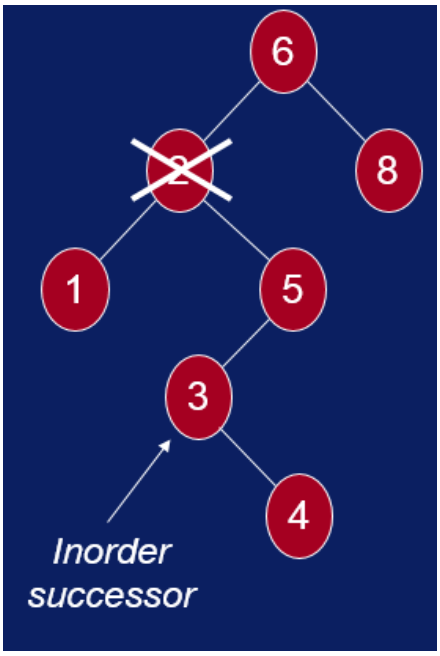
The inorder traversal order has to be maintained after the delete.



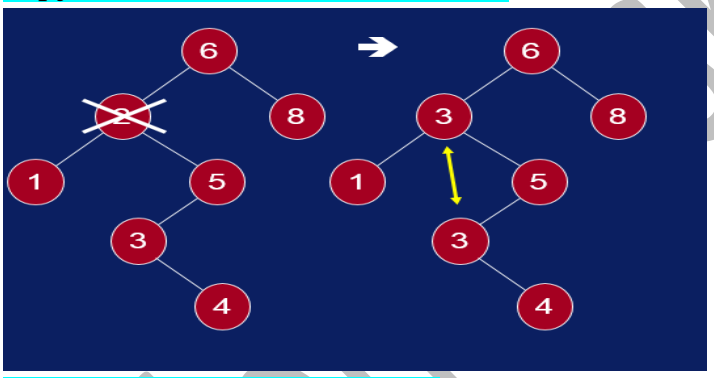
- The complicated case is when the node to be deleted has both left and right subtrees.
- The strategy is to replace the data of this node with the smallest data of the right subtree and recursively delete that node.

locate inorder successor

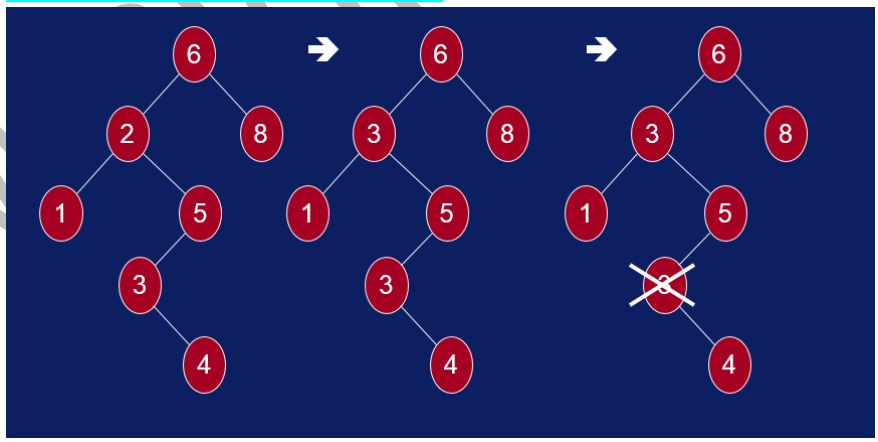
- Inorder successor will be the left-most node in the right subtree of 2.
- The inorder successor will not have a left child because if it did, that child would be the left-most node.



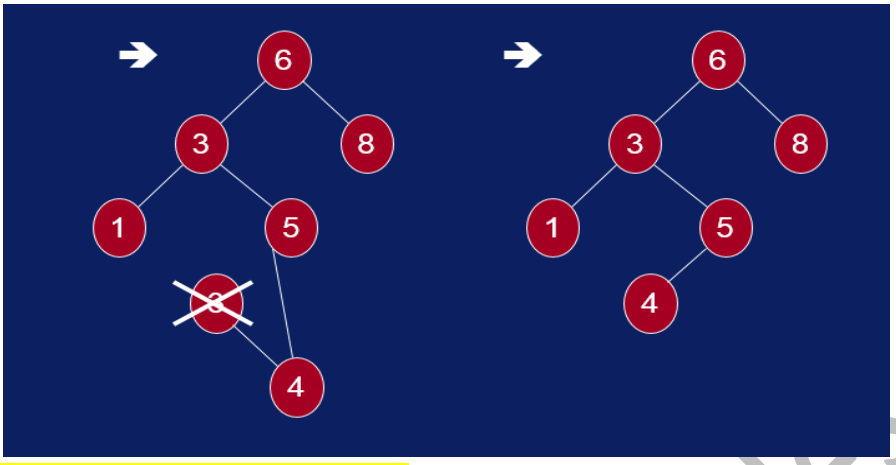
copy data from inorder successor.



remove the inorder successor



'delete' is C++ keyword. We will call our deleteNode routine remove.



Here is the C++ code for remove.

C++ code for remove

```
TreeNode<int>* remove(TreeNode<int>* tree, int info)
{
   TreeNode<int>* t;
   int cmp = info - *(tree->getInfo());
   if( cmp < 0 ){
      t = remove(tree->getLeft(), info);
      tree->setLeft( t );
   }
   else if( cmp > 0 ){
      t = remove(tree->getRight(), info);
      tree->setRight( t );
   }
//two children, replace with inorder successor
   else if(tree->getLeft() != NULL
         && tree->getRight() != NULL ){
      TreeNode<int>* minNode;
      minNode = findMin(tree->getRight());
      tree->setInfo( minNode->getInfo() );
      t = remove(tree->getRight(),
               *(minNode->getInfo()));
      tree->setRight( t );
   }
```

vu fion musk files

```
else {  // case 1
        TreeNode<int>* nodeToDelete = tree;
        if( tree->getLeft() == NULL ) //will handle 0 children
           tree = tree->getRight();
        else if( tree->getRight() == NULL )
           tree = tree->getLeft();
        else tree = NULL;

        delete nodeToDelete;
    }
    return tree;
}
```

Find the minimum node in a tree

```
TreeNode<int>* findMin(TreeNode<int>* tree)
{
    if( tree == NULL )
        return NULL;
    if( tree->getLeft() == NULL )
        return tree; // this is it.
    return findMin( tree->getLeft() );
}
```

BinarySearchTree.h

```
#ifndef _BINARY_SEARCH_TREE_H_
#define _BINARY_SEARCH_TREE_H_
#include <iostream.h>       // For NULL
// Binary node and forward declaration
template <class EType>
class BinarySearchTree;
```

Declaration of binary node and the binary search tree

vu elon musk files

```
#ifndef _BINARY_SEARCH_TREE_H_
#define _BINARY_SEARCH_TREE_H_
#include <iostream.h>       // For NULL
// Binary node and forward declaration
template <class EType>
class BinarySearchTree;
template <class EType>
class BinaryNode
{
   EType element;
   BinaryNode *left;
   BinaryNode *right;
   BinaryNode( const EType & theElement,
               BinaryNode *lt, BinaryNode *rt )
     : element( theElement ), left( lt ),
       right( rt ) { }
   friend class BinarySearchTree<EType>;
};
template <class EType>
template <class EType>
class BinarySearchTree
{
public:
   BinarySearchTree( const EType& notFound );
   BinarySearchTree( const BinarySearchTree& rhs );
   ~BinarySearchTree( );
   const EType& findMin( ) const;
   const EType& findMax( ) const;
   const EType& find( const EType & x ) const;
   bool isEmpty( ) const;
   void printInorder( ) const;
void insert( const EType& x );
   void remove( const EType& x );
   const BinarySearchTree & operator=
            ( const BinarySearchTree & rhs );
```

vu elon musk files

Internal method to print a subtree rooted at t in sorted order.

```
    private:
   BinaryNode<EType>* root;
   // ITEM_NOT_FOUND object used to signal failed finds
   const EType ITEM_NOT_FOUND;
   const EType& elementAt( BinaryNode<EType>* t );
   void insert(const EType& x, BinaryNode<EType>* & t);
   void remove(const EType& x, BinaryNode<EType>* & t);
   BinaryNode<EType>* findMin(BinaryNode<EType>* t);
   BinaryNode<EType>* findMax(BinaryNode<EType>* t);
   BinaryNode<EType>* find(const EType& x,
                           BinaryNode<EType>* t );
   void makeEmpty(BinaryNode<EType>* & t);
   void printInorder(BinaryNode<EType>* t);
};
#endif
```

Lecture No. 17

Reference Variables

- The symbol "&" has a few different purposes depending on where it occurs in code.When it appears in front of a variable name, it is the address operator, i.e., it returns the address of the variable in memory.

```
int x;
int* ptr = &x;
```

- The symbol "&' can also appear after a type in a function signature:

For example, insert and remove from the BinarySearchTree class.

```
void insert( const EType& x );

void remove( const EType& x );
```

vu Elon musk files

- Or, in case we designed the BinarySearchTree class to hold integers only, i.e., no templates

```
void insert( const int& x );
void remove( const int& x );
```

- The "&" indicates a parameter that is a *reference variable*.

Consider the following three different functions:

```
// example 1
int intMinus1( int oldVal)
{
        oldVal = oldVal – 1;
        return oldVal;
}
// example 2
int intMinus2( int* oldVal)
{
        *oldVal = *oldVal – 2;
        return *oldVal;
}
// example 3
int intMinus3( int& oldVal)
{
        oldVal = oldVal – 3;
        return oldVal;
```

vu Elon musk files

```
}
```

The caller function: calling intMinus1

```
void caller()
{
        int myInt = 31;
        int retVal;
        retVal = intMinus1( myInt );
        cout << myInt << retVal;
}
```

Reference Variables

- Suppose we want a function to change an object.
- But we don't want to send the function a copy. The object could be large and copying it costs time.
- We don't want to use pointers because of the messy syntax.

call-by-reference (or pass-by-reference)

```
void caller()
{
        int retVal;
        int myInt = 31;
        retVal = intMinus3( myInt );
        cout << myInt << retVal;
}
```

The & after int means that oldVal is an integer reference variable.

vu Elon musk files

```
// example 3
int intMinus3( int& oldVal)
{
        oldVal = oldVal – 3;
        return oldVal;
}
```

So what *is* a reference variable?

- The idea is: the integer object myInt is used exactly as it exists in the caller. The function simply reaches it through a *different name*, oldVal.
- The function intMinus3 cannot use the name myInt because it is in the caller's scope.
- But both variable names refer to the same object (same memory cell).

vu lion musk files

Lecture No. 18

Reference Variables

- One should be careful about transient objects that are stored by reference in data structures.
- Consider the following code that stores and retrieves objects in a queue.

stores and retrieves objects in a queue

```
void loadCustomer( Queue& q)
{
```

```
        Customer c1("irfan");
        Customer c2("sohail";
        q.enqueue( c1 );
        q.enqueue( c2 );
}
```

serviceCustomer() method

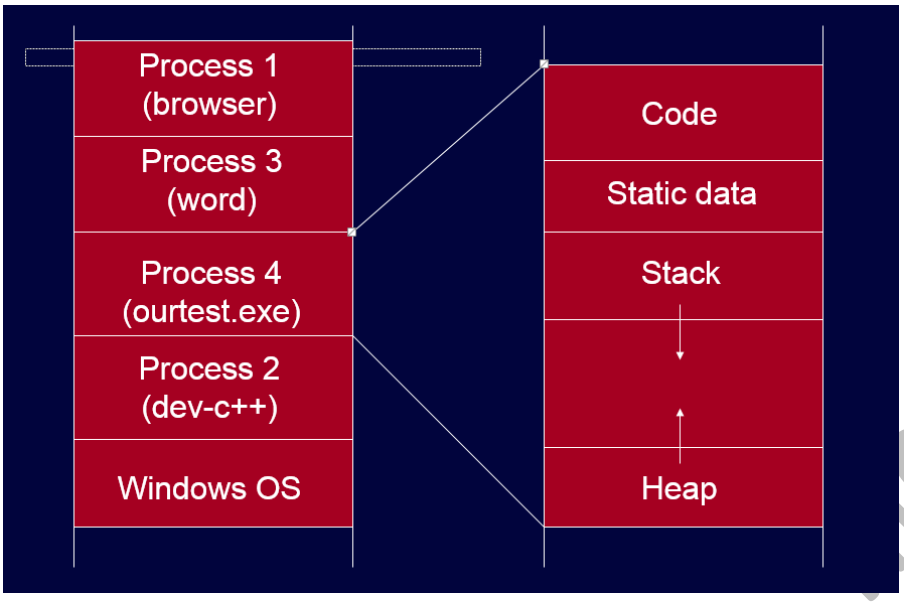
```
void loadCustomer( Queue& q)
{
        Customer* c1 = new Customer("irfan");
        Customer* c2 = new Customer("sohail";
        q.enqueue( c1 ); // enqueue takes pointers
        q.enqueue( c2 );
}
```

- The pointer variables c1 and c2 are on the call stack. They will go but their contents (addresses) are queued.
- The Customer objects are created in the heap. They will live until explicitly deleted.

Memory Organization

There is another heap data structure is an area in memory given to a process to operating system for dynamic memory allocation.

vu elon musk files

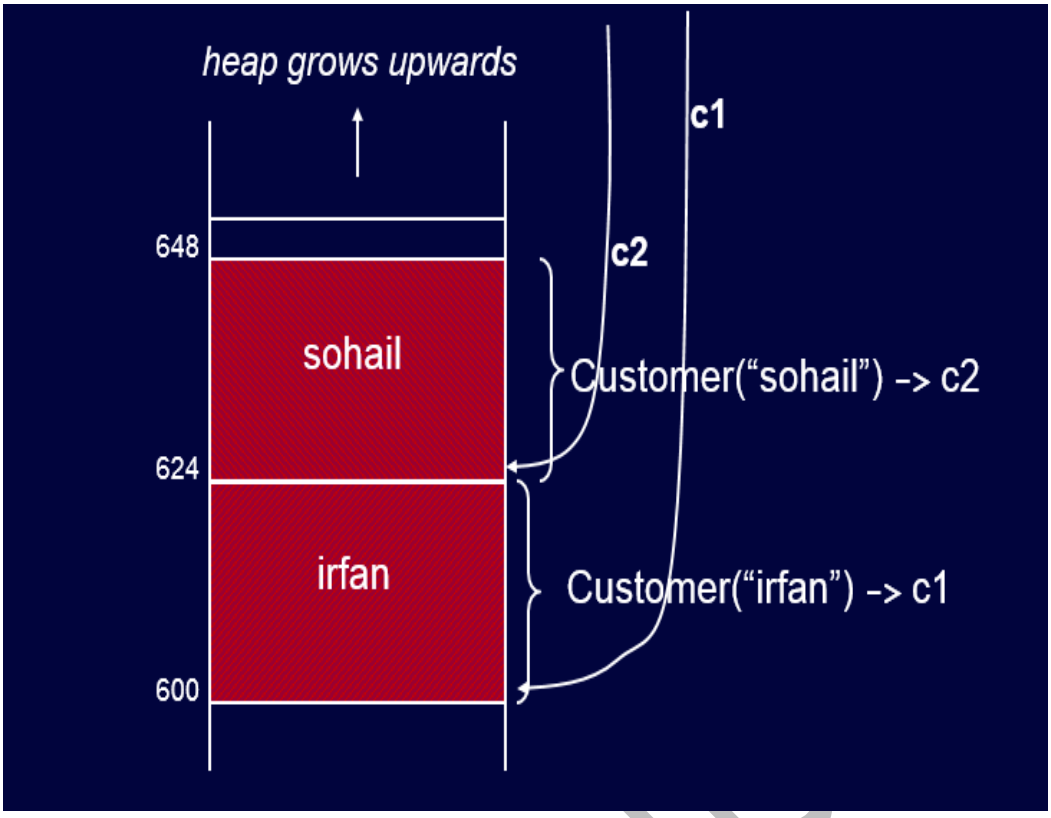


One the left of the picture, we can see different processes in the computer memory. When we zoomed into the one of the processes, we saw the picture on the right. That firstly, there is a section for code, then for static data and for stack. Stack grows in the downward section. You can see the heap section given at the end, which grows upward. An interesting question arises here is that why the stack grows downward and heap in the upward direction. Think about an endless recursive call of a function to itself. For every invocation, there will be an activation record on stack. So the stack keeps on growing and growing even it overwrites the heap section. One the other hand, if your program is performing dynamic memory allocation endlessly, the heap grows in the upward direction such that it overwrites the stack section and destroys it.

if a process has some destructive code then it will not harm any other process, only its own destruction is caused. By the way, lot of viruses exploit the stack overflow to change the memory contents and cause further destruction to the system.

let's see the figure below where the objects are created dynamically.

www.elonmusk.es



Reference Variables

```
void serviceCustomer( Queue& q)
{
        Customer* c = q.dequeue();
        cout << c->getName() << endl;
        delete c; // the object in heap dies
}
```

Must use the c-> syntax because we get a pointer from the queue.The object is still alive because it was created in the heap.

The const Keyword

- *Use 1*: The const keyword appears before a function parameter. E.g., in a chess program:

vu Elon mus

int movePiece(const Piece& currentPiece)

- The parameter must remain constant for the life of the function.
- If you try to change the value, e.g., parameter appears on the left hand side of an assignment, the compiler will generate and error.
- This also means that if the parameter is passed to another function, that function must not change it either.
- Use of const with reference parameters is very common.
- This is puzzling; why are we passing something by reference and then make it constant, i.e., don't change it?
- Doesn't passing by reference mean we *want* to change it?

- The answer is that, yes, we don't want the function to change the parameter, but neither do we want to use up time and memory creating and storing an entire copy of it.
- So, we make the original object available to the called function by using pass-by-reference.
- We also mark it constant so that the function will not alter it, even by mistake.

www.elonmusk.files

Lecture No. 19

The const Keyword

Use 2: The const keyword appears at the end of class member's function signature.

```
EType& findMin( ) const;
```

- Such a function cannot change or write to member variables of that class.

- This type of usage often appears in functions that are suppose to read and return member variables.

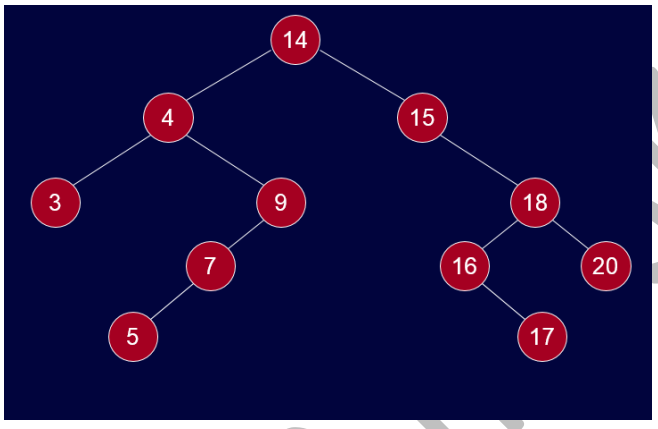
  Use 3: The const keyword appears at the beginning of the return type in function signature:

  const EType& findMin( ) const;

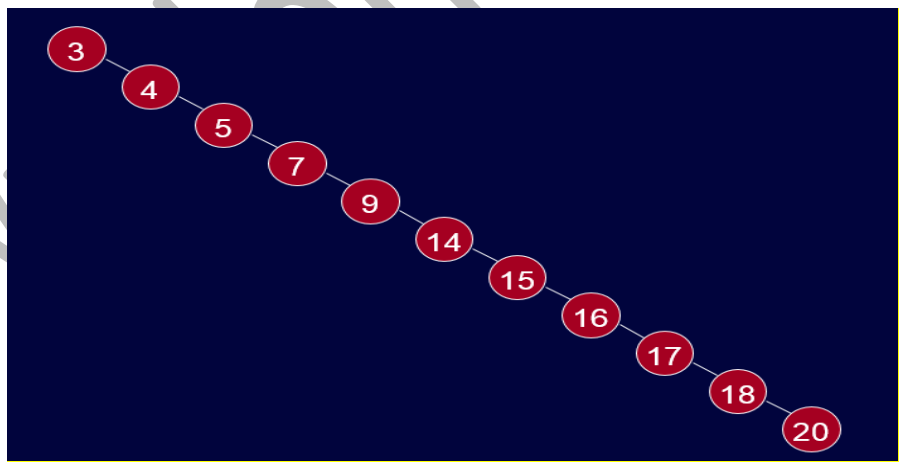
- Means, whatever is returned is constant.
- The purpose is typically to protect a reference variable.
- This also avoids returning a copy of an object.
-

Degenerate Binary Search Tree

BST for 14, 15, 4, 9, 7, 18, 3, 5, 16, 20, 17



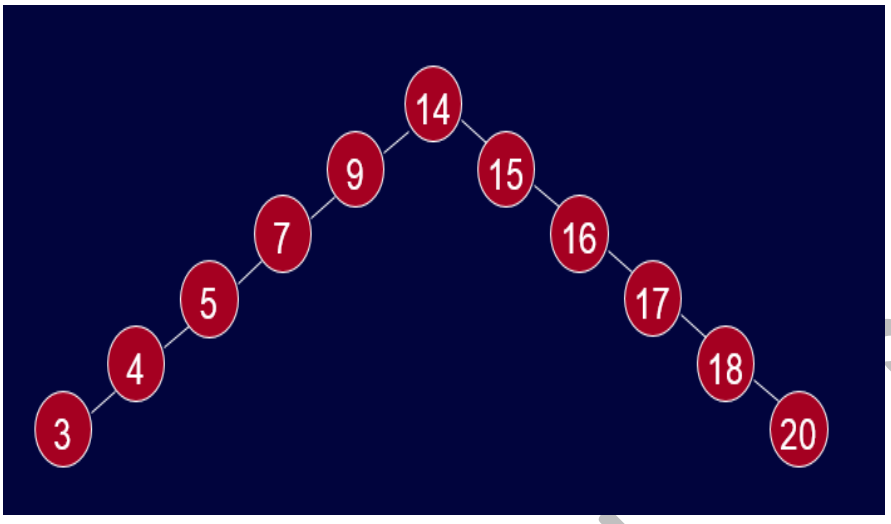
BST for 3 4 5 7 9 14 15 16 17 18 20



Balanced BST

- We should keep the tree *balanced*.

- One idea would be to have the left and right subtrees have the same height. Balanced BST Does not force the tree to be shallow:

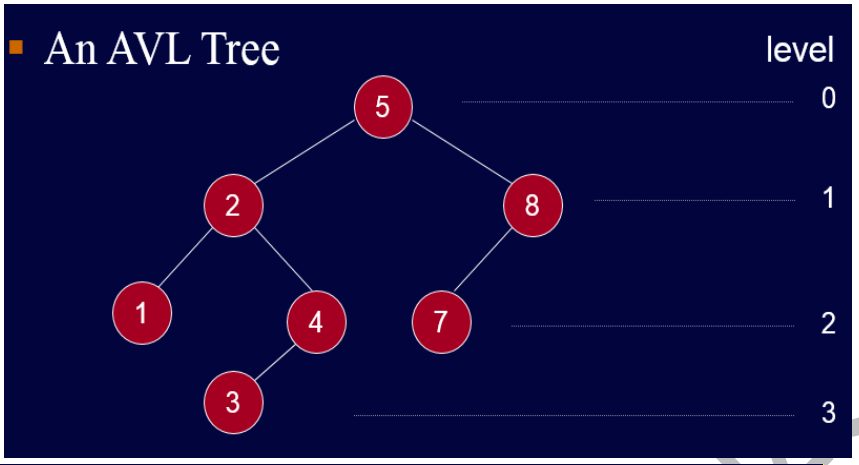


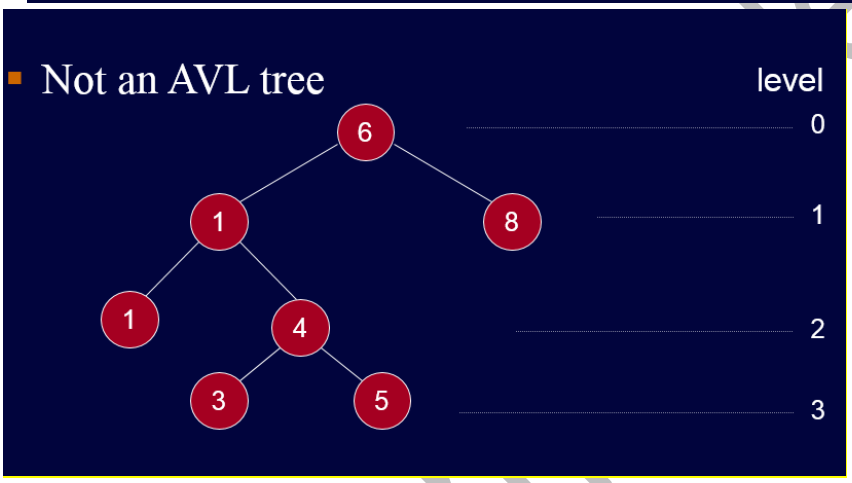
- We could insist that every node must have left and right subtrees of same height.
- But this requires that the tree be a complete binary tree
- To do this, there must have $(2^{d+1} - 1)$ data items, where d is the depth of the tree.
  This is too rigid a condition.

AVL Tree

- AVL (Adelson-Velskii and Landis) tree.
- An AVL tree is identical to a BST except
  - height of the left and right subtrees can differ by at most 1.
  - height of an empty tree is defined to be (–1).

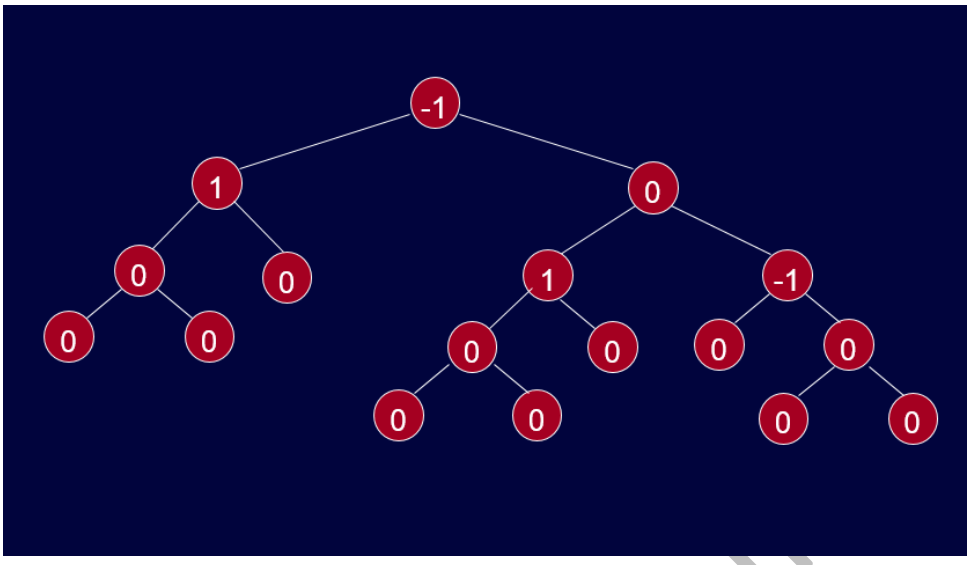
vu Elon musk





Balanced Binary Tree

1. The *height* of a binary tree is the maximum level of its leaves (also called the depth).
2. The balance of a node in a binary tree is defined as the height of its left subtree minus height of its right subtree.
3. Here, for example, is a balanced tree. Each node has an indicated balance of 1, 0, or –1.



Lecture No. 20

AVL Tree

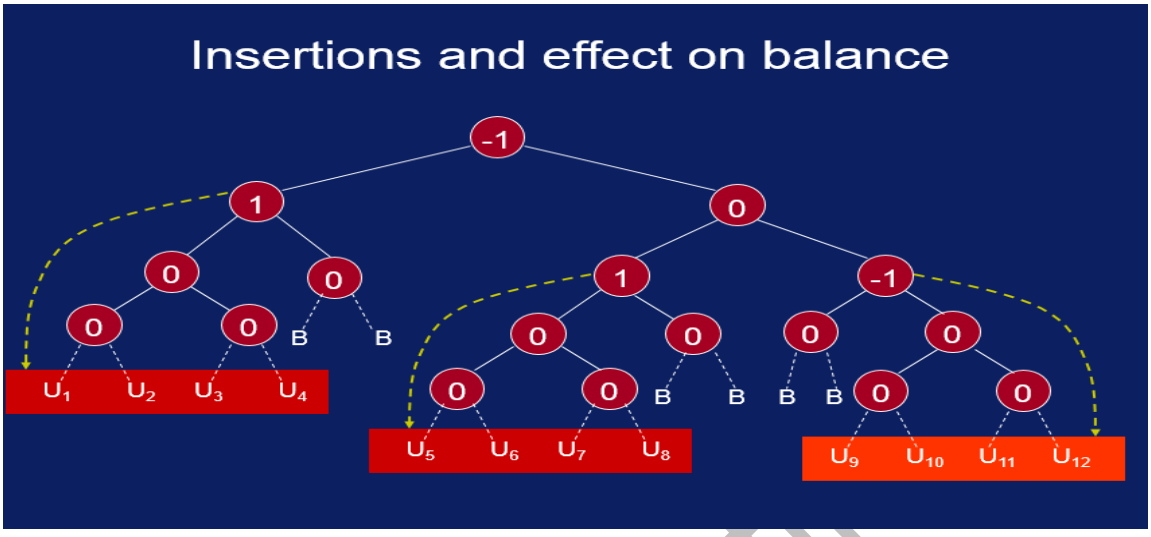
In the year 1962, two Russian scientists, Adelson-Velskii and Landis, proposed the criteria to save the binary search tree (BST) from its degenerate form. This was an effort to propose the development of a balanced search tree by considering the height as a standard. This tree is known as AVL tree. The name AVL is an acronym of the names of these two scientists.

Balanced Binary Tree insertion

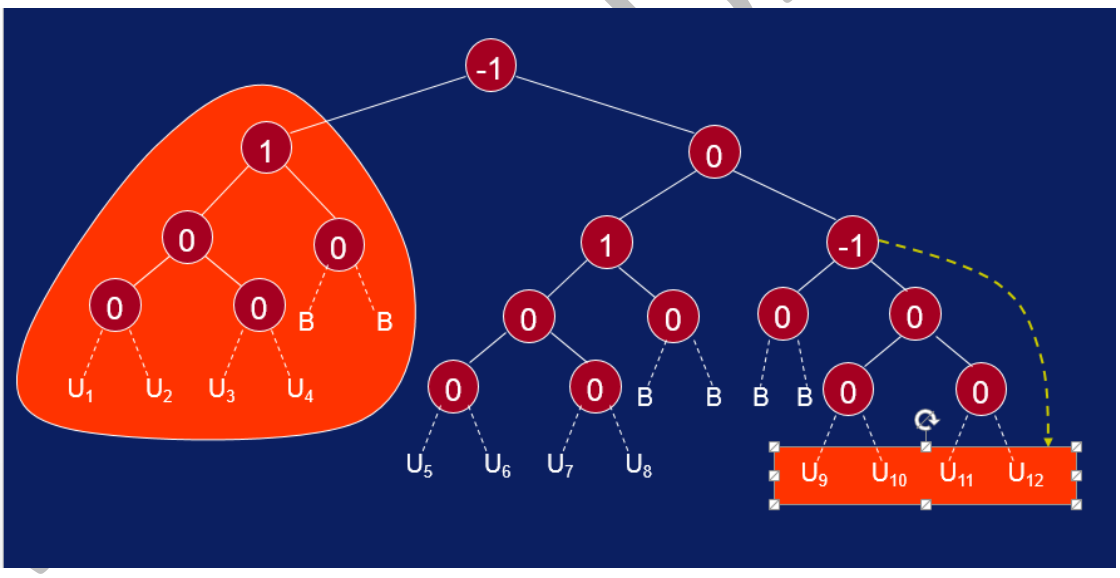
Tree becomes unbalanced only if the newly inserted node.

- Is a left descendant of a node that previously had a balance of 1 (U_1 to U_8),
- Is a descendant of a node that previously had a balance of –1 (U_9 to U_{12}).

vu elon musk



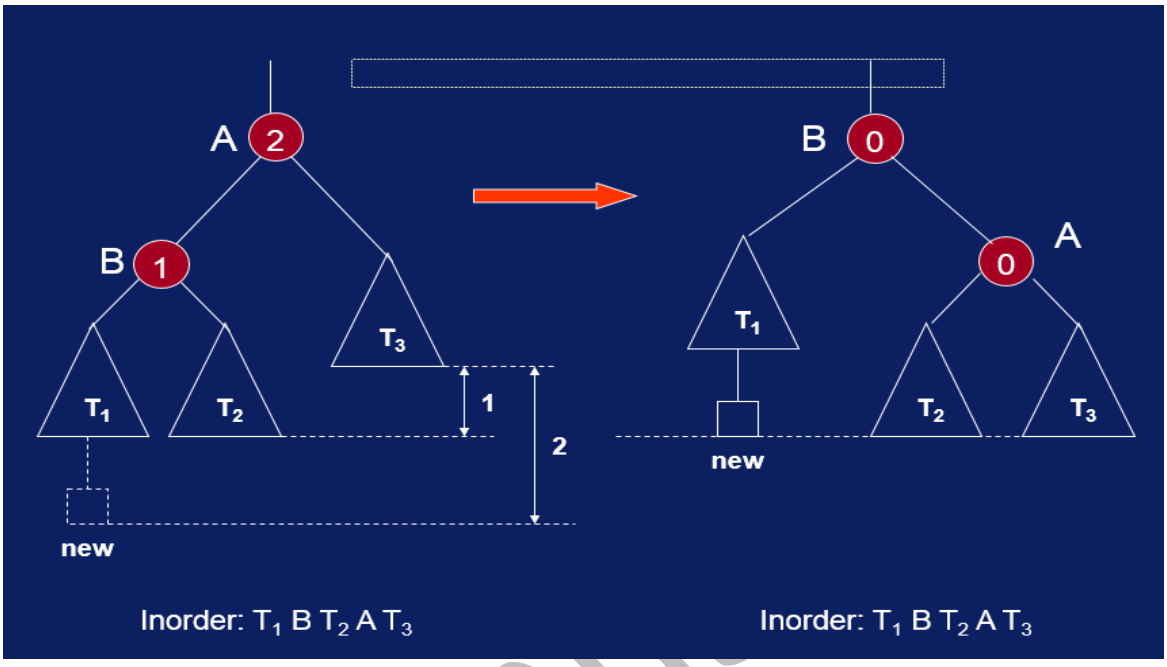
Consider the case of node that was previously 1



Insertion of Node in an AVL Tree

To insert a node in a BST, we compare its data with the root node. If the new data item is less than the root node item in a particular order, this data item will hold its place in the left subtree of the root. Now we compare the new data item with the root of this left subtree and decide its place. Thus at last, the new data item becomes a leaf node at a proper place. After inserting

the new data item, if we traverse the tree with the inorder traversal, then that data item will become at its appropriate position in the data items.



AVL Tree Building Example

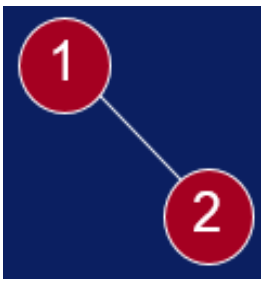
inserts numbers in a balanced search tree.

We will check the balance after each insert and rebalance if necessary using rotations.

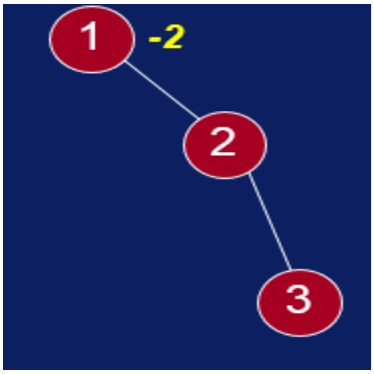
Inser t(1)



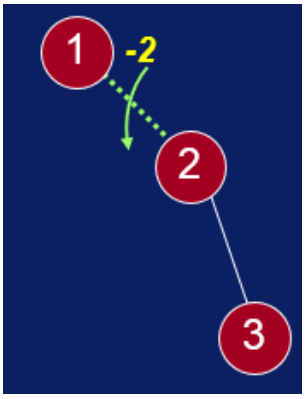
Insert (2)



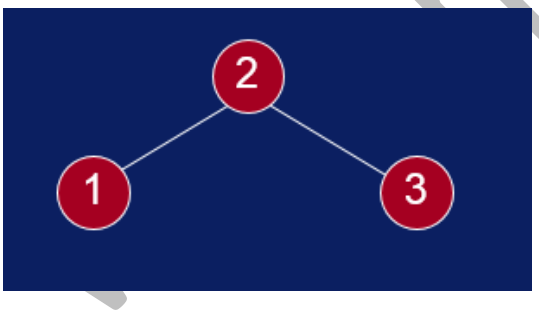
Insert (3) single left rotation



Insert (3) single left rotation



Insert(3)



So we can see that after the rotation, the tree has become balanced. The figure reflects that the balance of node 1, 2 and 3 is 0. We see that the inorder traversal of the above tree before rotation (tree on left hand side) is 1 2 3. Now if we traverse the tree after rotation (tree on right hand side) by in order traversal, it is also 1 2 3. With respect to the inorder traversal, both the traversals are same.

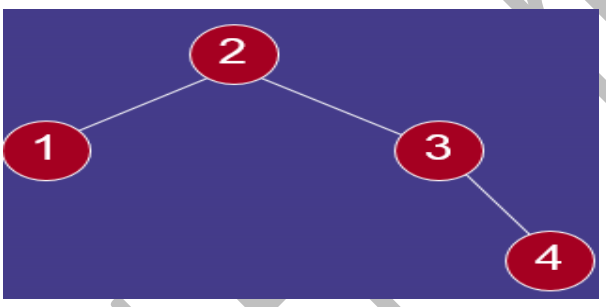
Observe that the position of nodes in a tree does not matter as long as the inorder traversal remains the same. We have seen this in the above figure where two different trees give the same inorder traversal. In the same way we can insert more nodes to the tree. After inserting a node we will check the balance of nodes whether it violates the AVL condition. If the tree, after inserting a node, becomes unbalance then we will apply rotation to make it balance. In this way we can build a tree of any number of nodes.

Lecture No. 21

AVL Tree Building

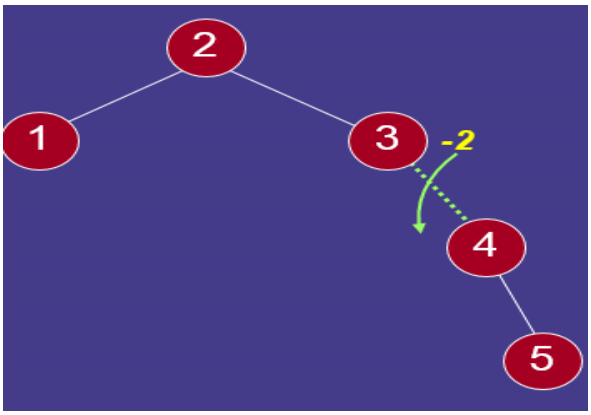
Let's insert few more nodes in the tree. We will build an AVL tree and rotate the node when required to fulfill the conditions of an AVL tree.

To insert a node containing number 4, we will, at first, compare the number inside the Root node. The current root node is containing number 2. As 4 is greater than 2, it will take the right side of the root. In the right subtree of the root, there is the node containing number 3. As 4 is also greater than 3, it will become the right child of the node containing number 3.

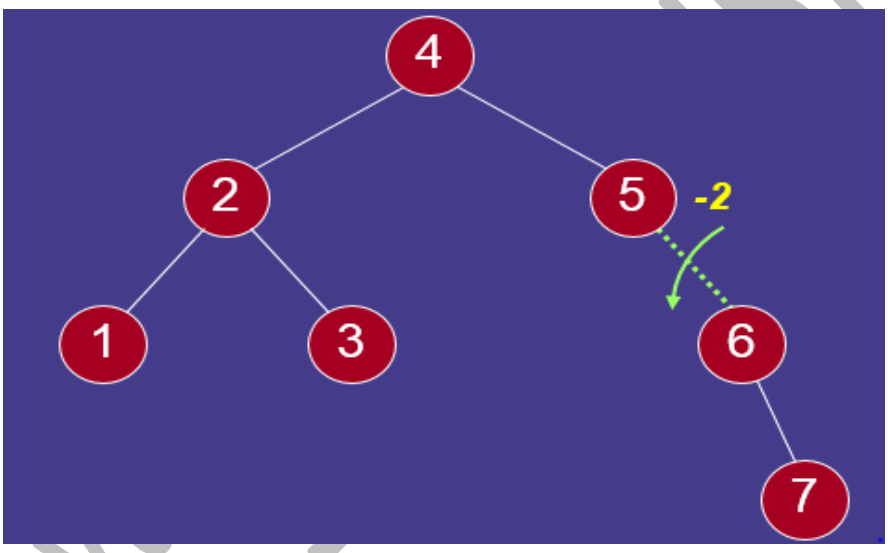


Once we insert a node in the tree, it is necessary to check its balance to see whether it is within AVL defined balance. If it is not so, then we have to rotate a node. If the balance factor of each node inside the tree is 0, it will be a perfectly balanced tree.

www.t.me/muskfiles



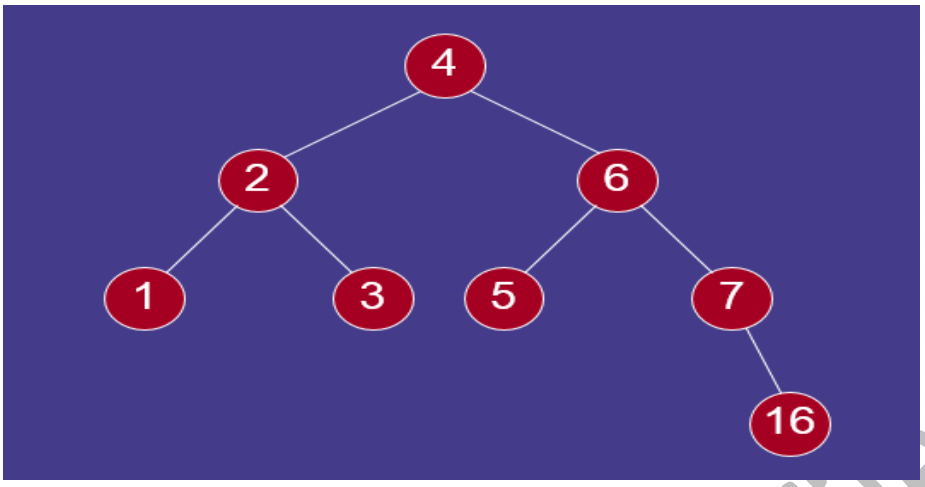
Now, we have inserted a node containing number 5 and see the balance factor of each node. The balance factor for the node containing 5 is 0. The balance factor for node containing 4 is –1 and for the node containing 3 is -2. The condition for AVL is not satisfied here for the node containing number 3, as its balance factor is –2. The rotation operation will be performed here as with the help of an arrow.



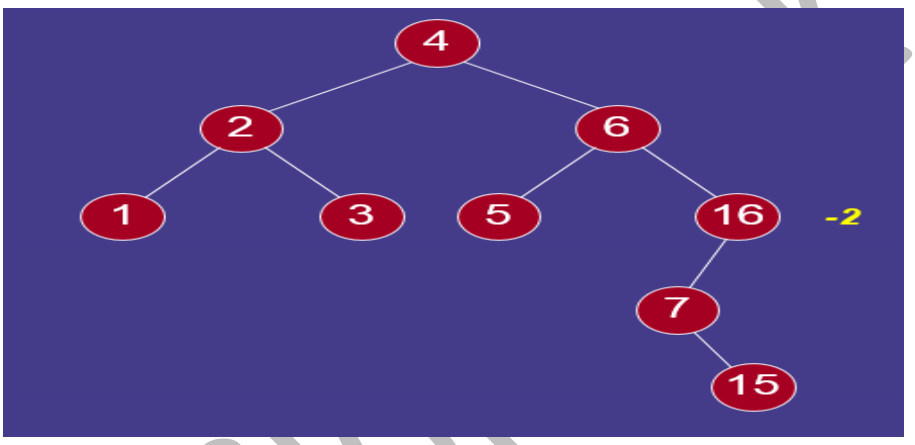
Let' see the inorder traversal output here:

1 2 3 4 5 6 7

It is still in the same sequence and the number 7 has been added at the end. Now insert a new node 16 in the tree.



Now, let's compute the balance factors for the nodes. The balance factor for nodes 16, 7, 5, 3, 1, 6, 2 and 4 is either 0 or –1. So this fulfills the condition of a tree to be an AVL. Let's insert another node containing number 15 in this tree.



Now node 7 has become the left child of node 16 while node 15 has attained the form of the right child of node 7. Now the balance factors for node 15, 7 and 16 are 0, -1 and 2 respectively. Note that the single rotation above when we rotated node 7 is not enough as our tree is still not an AVL one.

Cases for Rotation

- Single rotation does not seem to restore the balance.
- The problem is the node 15 is in an inner subtree that is too deep.

Let us revisit the rotationsLet us call the node that must be rebalanced α.

Since any node has at most two children, and a height imbalance requires that α's two subtrees differ by two (or –2), the violation will occur in four cases.

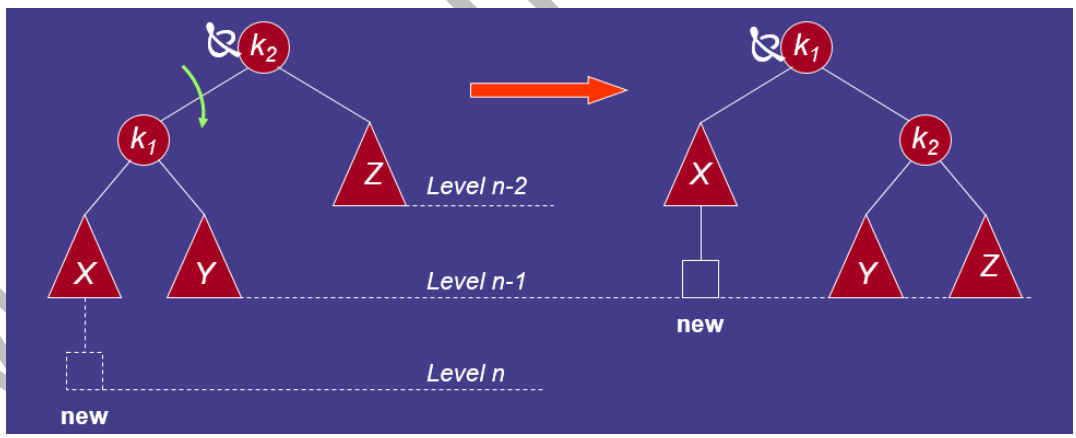
- An insertion into left subtree of the left child of α.
- An insertion into right subtree of the left child of α.
- An insertion into left subtree of the right child of α.
- An insertion into right subtree of the right child of α.

The insertion occurs on the "outside" (i.e., left-left or right-right) in cases 1 and 4

Single rotation can fix the balance in cases 1 and 4.

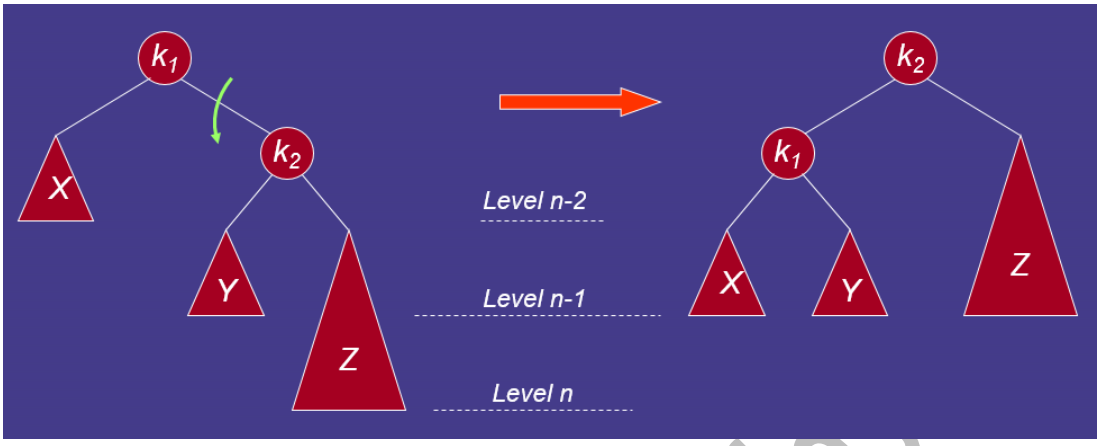
Insertion occurs on the "inside" in cases 2 and 3 which single rotation cannot fix.

Single right rotation to fix case 1.

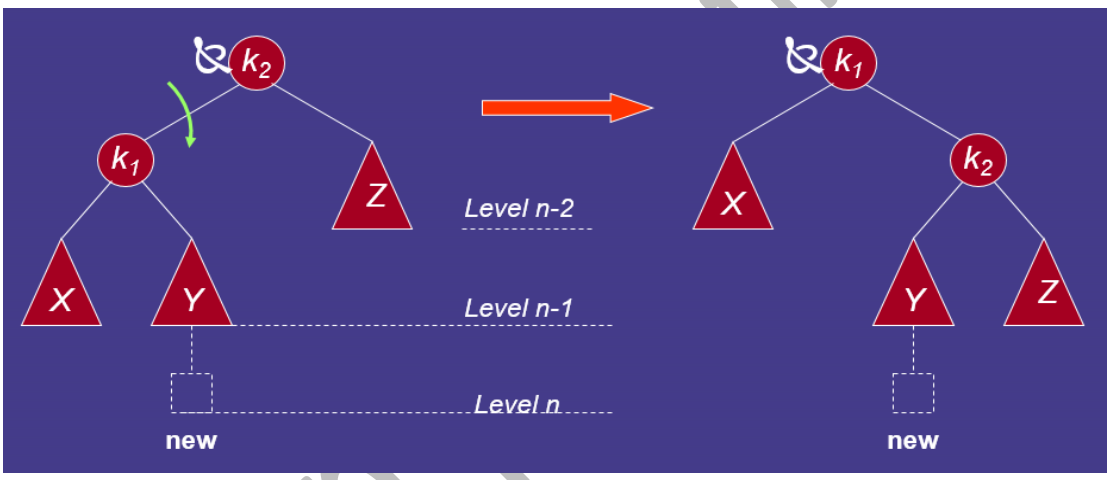


Single left rotation to fix case 4.

www.anusk files



Single right rotation *fails* to fix case 2



This is an inside insertion. The balance factor for the node k2 became 2. We make single rotation by making right rotation on the node k2 as shown in the figure on the right. We compute the balance factor for k1, which is –2. So the tree is still not within the limits of AVL tree. Primarily the reason for this failure is the node Y subtree, which is unchanged even after making one rotation. It changes its parent node but its subtree remains intact.

Lecture No. 22

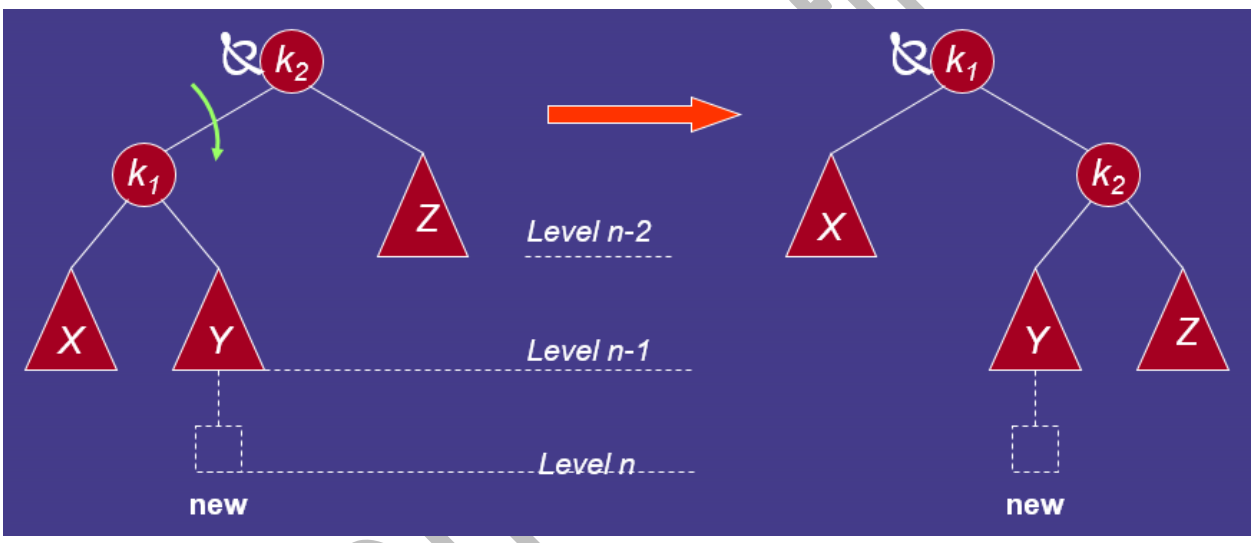
Cases of rotations

As We have analyzed the insertion method again and talked about the α node. The new node will be inserted at the left or right subtree of the α's left child or at the

left or right subtree of the α's right child. Now the question arises whether the single rotation help us in balancing the tree or not.

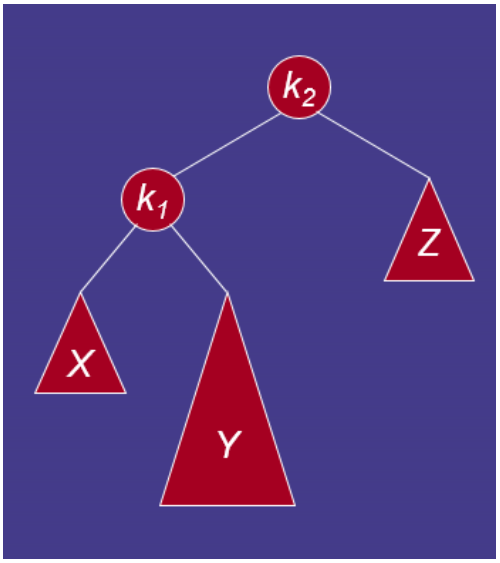
If the new node is inserted in the left subtree of the α's left child or in the right subtree of α's right child, the balance will be restored through single rotation. However, if the new node goes inside the tree, the single rotation is not going to be successful in balancing the tree. We face four scenarios in this case. We said that in the case-1 and case-4, single rotation is successful while in the case-2 and case-3 single rotation does not work.

Single right rotation *fails* to fix case 2

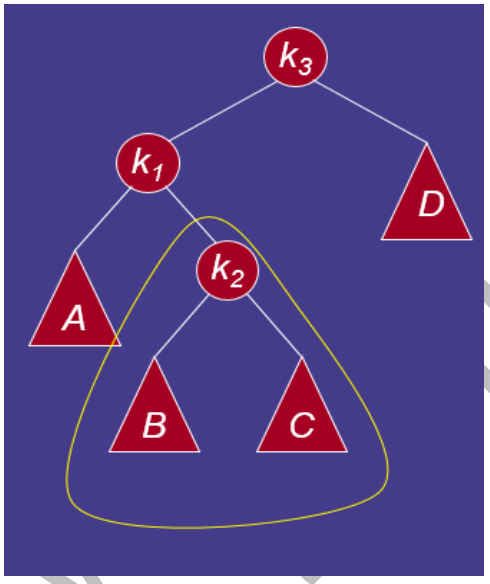


In the above tree, we have α node as k2, which has a left child as k1. Whereas X and Y are its left and right children. The node k2 has a right child Z. Here the newly inserted node works as the left or right child of node Y. Due to his insertion, one level is increased in the tree. We have applied single rotation on the link of k1 and k2. The right side tree in the figure is the post-rotation tree. The node k1 is now at the top while k2 comes down and node Y changes its position. Now if you see the levels of the node, these are seen same. Have a look on the level of the α node i.e. k1 which reflects that the difference between the left and right side levels is still 2. So the single rotation does not work here.

View the entire tree with four subtrees connected with 3 nodes



- Y is non-empty because the new node was inserted in Y.
- Thus Y has a root and two subtrees.

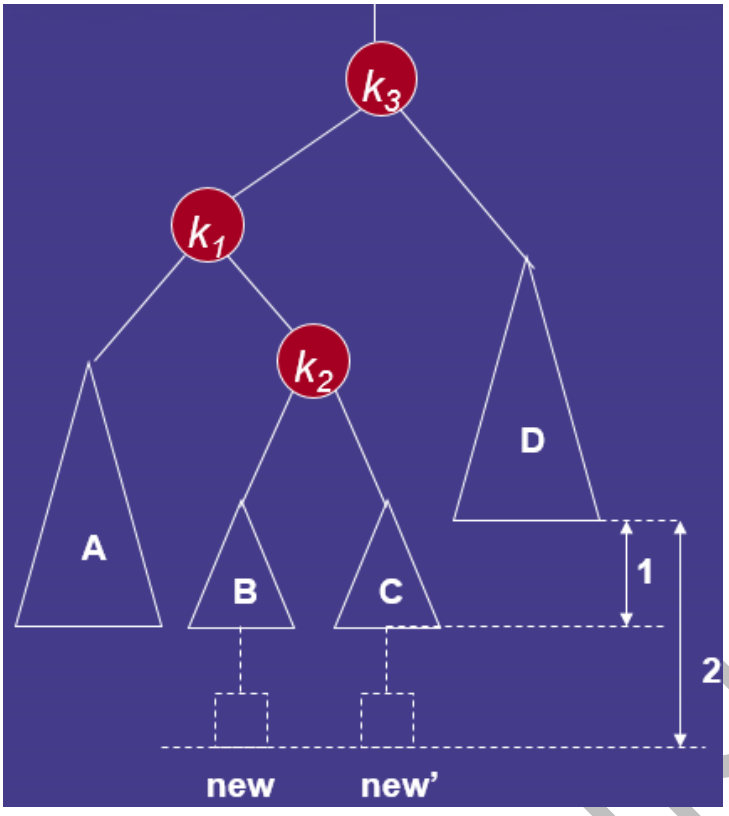


Here exactly one of tree *B* or *C* is two levels deeper than *D*; we are not sure which one.

- Good thing: it does not matter.
- To rebalance, k_3 cannot be left as the root.

New node inserted a either of the two spots

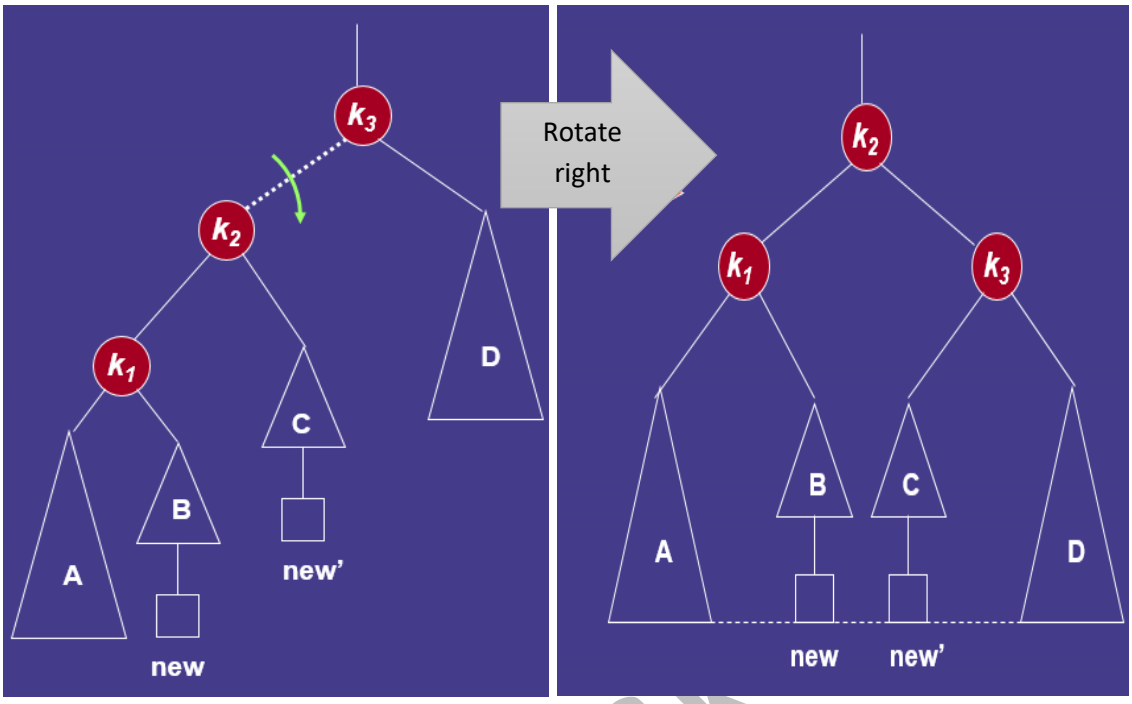
musk files



- A rotation between k_3 and k_1 (k_3 was k_2 then) was shown to not work.
- The only alternative is to place k_2 as the new root.
- This forces k_1 to be k_2's left child and k_3 to be its right child.

Left-right *double* rotation to fix case 2

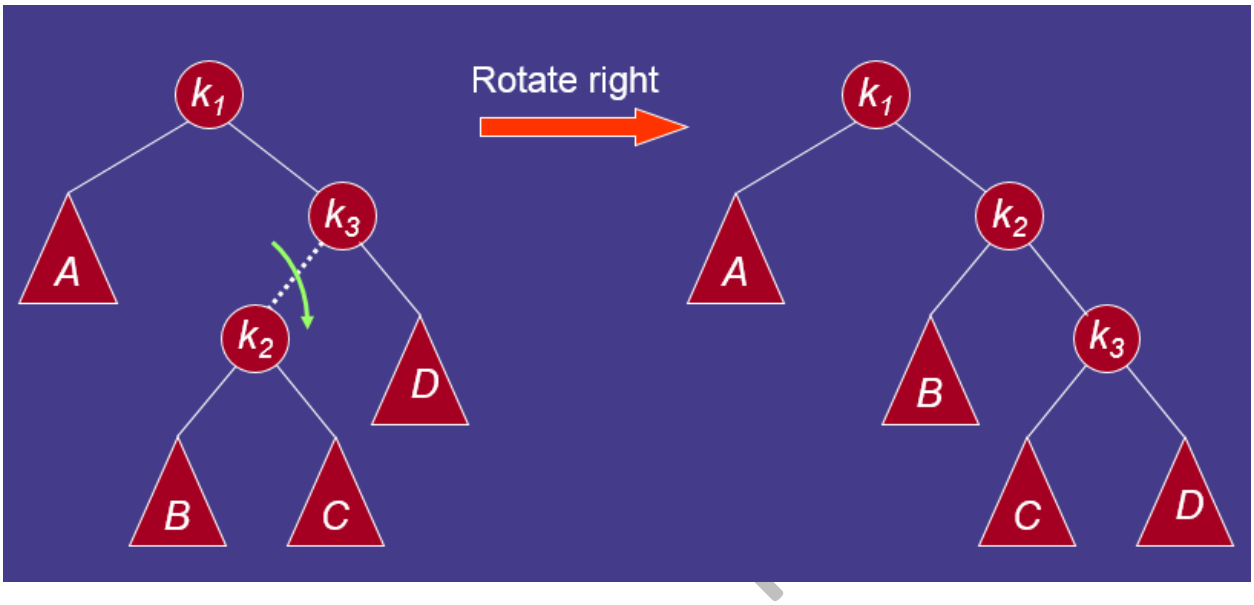
vu ElonMusk files



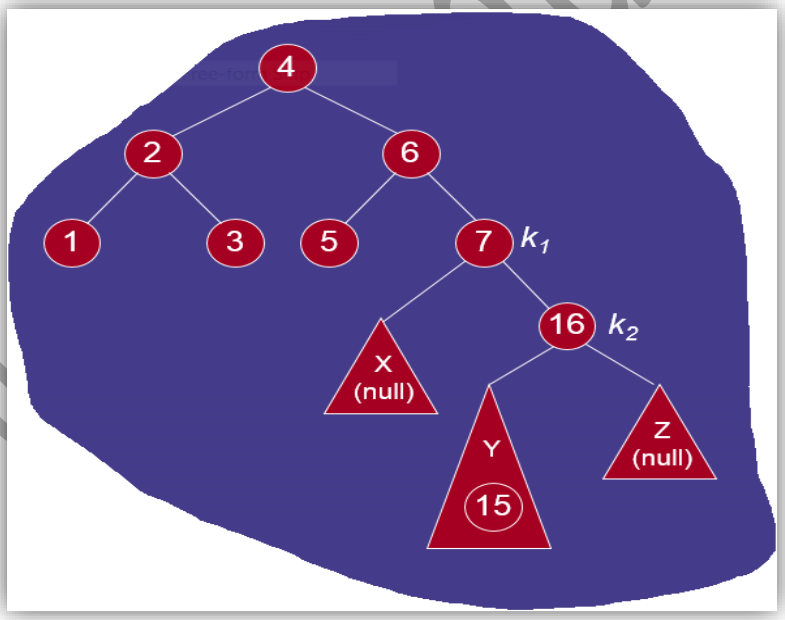
Now, we have the final shape of the tree. You can see that *k2* has become the root of the tree. *k1* and *k3* are its left and right children respectively. While performing the inorder traversal, you can see that we have preserved our inorder traversal. We inserted numbers in it.

When the balance factor becomes more than one, rotation is performed. During this process, we came at a point when single rotation failed to balance the tree. Now there is need to perform double rotation to balance the tree that is actually two single rotations. Do not take double rotation as some complex function, it is simply two single rotations in a special order. This order depends on the final position of the new node. Either the new node is inserted at the right subtree of the left child of α node or at the left subtree of the right child of α node. In first case, we have to perform left right rotation while in the second case, the right-left rotation will be carried out.

.

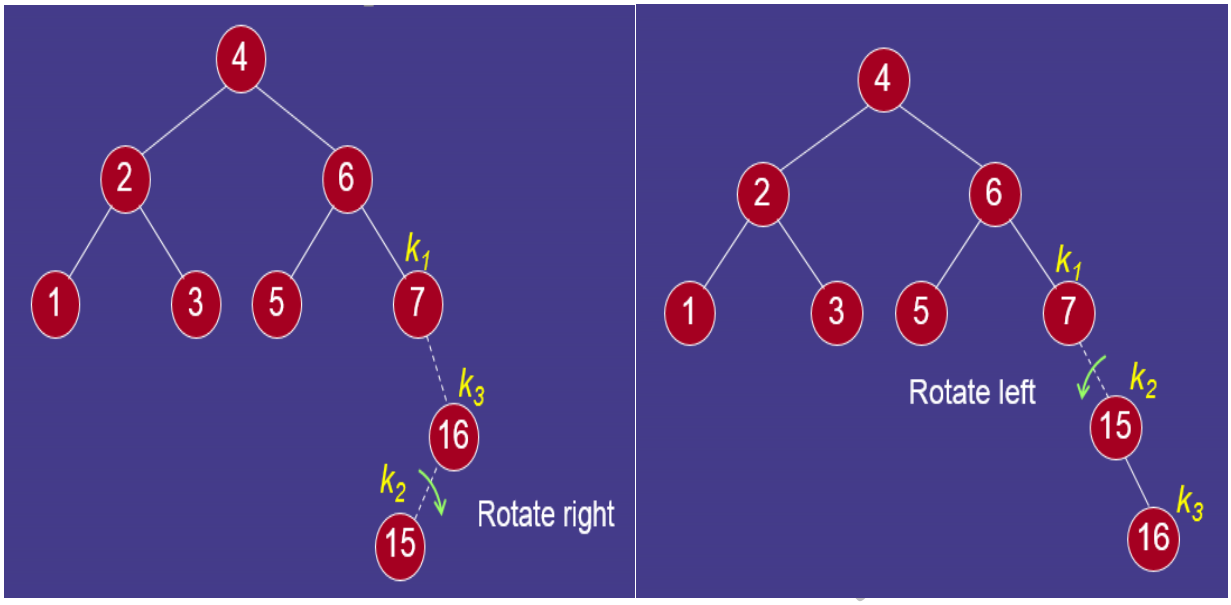


Let's go back to our example and try to complete it. So far, we have 1, 2, 3, 4, 5, 6, 7 and 16 in the tree and inserted 15 which becomes the left child of the node 16.



Right-left double rotation:

Here we have shown *X*, *Y* and *Z* in case of the double rotation. We have shown *Y* expanded and 15 is inside it. Here we will perform the double rotation, beginning with the right rotation first.



C++ Code for avlInsert method

Now we have to balance the procedure in the *insert* method. We have already written this *insert* method which takes some value and adds a node in the tree. That procedure does not perform balancing. Now we will include this balancing feature in our *insert* method so that the newly created tree fulfills the AVL condition.

This is the function used to insert nodes satisfying the AVL condition

```
TreeNode<int>*
avlInsert(TreeNode<int>* root, int info)
{
   if( info < root->getInfo() ){
      root->setLeft(avlInsert(root->getLeft(), info));
      int htdiff = height(root->getLeft()) –
                   height(root->getRight());
      if( htdiff == 2 )
         if( info < root->getLeft()->getInfo() )
           root = singleRightRotation( root );
         else
           root = doubleLeftRightRotation( root );
```

vu Elon musk

```
  }
else if(info > root->getInfo() ) {
       root->setRight(avlInsert(root->getRight(),info));
       int htdiff = height(root->getRight()) –
                height(root->getLeft());
       if( htdiff == 2 )
          if( info > root->getRight()->getInfo() )
             root = singleLeftRotation( root );
          else
             root = doubleRightLeftRotation( root );
    }
    // else a node with info is already in the tree. In
 // case, reset the height of this root node.
    int ht = Max(height(root->getLeft()),
                height(root->getRight()));
    root->setHeight( ht+1 ); // new height for root.
    return root;
}
```

The input arguments are *root* node and the *info* is of the type *int*. In our example, we are having a tree of *int* data type. But of course, we can have any other data type. The return type of *avlInsert* method is *TreeNode<int>**. This *info* will be inserted either in the right subtree or left subtree of *root*.

- The first *if* statement is making this decision by comparing the value of *info* and the value of root.
- Than we are calling *avlInsert* method again. We are passing it the first node of the left subtree and the *info* value.
- Than we are inserting the new node and also getting the root of the left subtree after the insertion of the new node.
- So after this, we check the balance factor of the node. We call a function *height* to get the height of left and right subtree of the root.
- After getting the heights of left and right subtrees, we take its difference.
- Now if the difference of the heights of left and right subtrees is greater than 1, we have to rebalance the tree.
- We will perform single rotation or double rotation to balance the tree.

vu fion musk files

- This rotation will be carried out with only one node where the balance factor
  is 2. As a result, the tree will be balanced.
- After this, we check the balance factor. We calculate the difference as:

  int htdiff = height(root->getRight()) – height(root->getLeft());

vu £lon musk files