Data Structures – Week #5

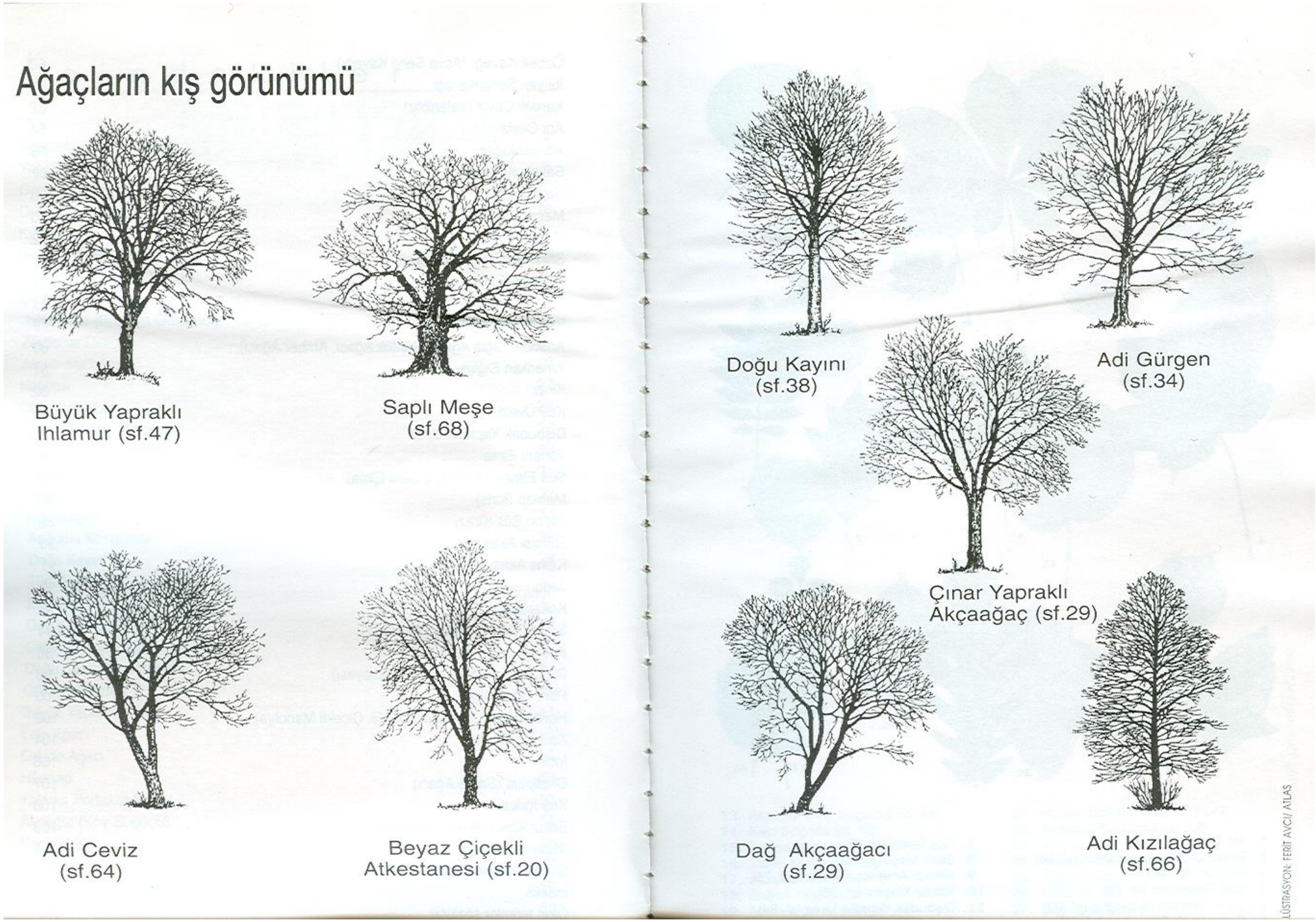
Trees (Ağaçlar)

Trees (Ağaçlar)

Toros Göknarı

Avrupa Göknarı

Trees (Ağaçlar)



Outline

- Trees
- Definitions
- Implementation of Trees
- Binary Trees
- Tree Traversals & Expression Trees
- Binary Search Trees

Definition of a Tree

- Definition:

  A ***tree*** is a collection of nodes (vertices) where the collection may be empty. Otherwise, the collection consists of a distinguished node *r*, called the ***root***, and zero or more non-empty (sub)trees T_1, …, T_k, each of whose roots are connected by a *directed edge* to *r*.

More definitions

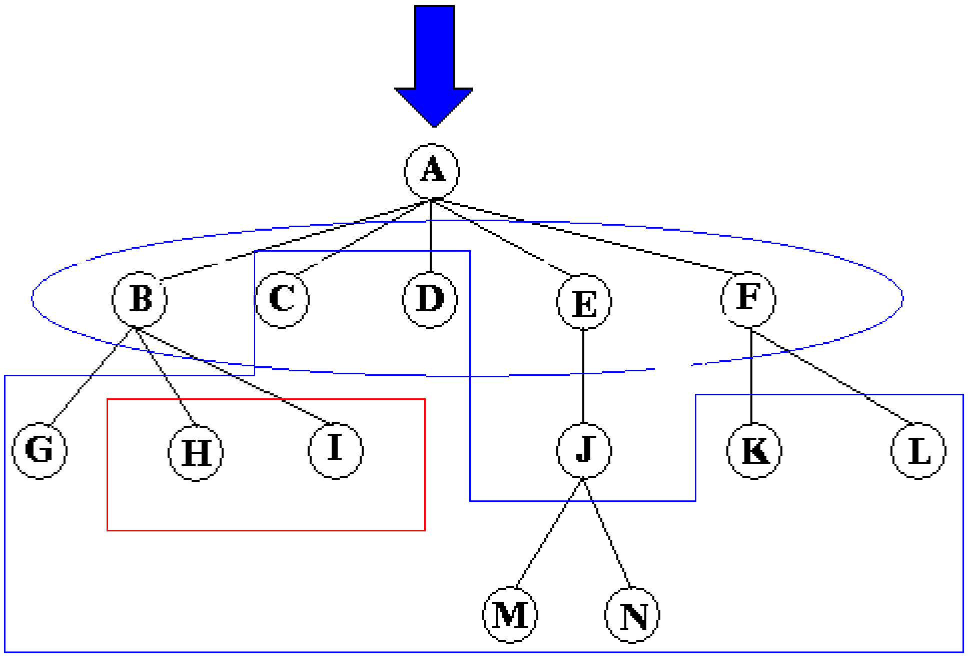
- ***In-degree*** of a vertex is the number of edges arriving at that vertex.
- ***Out-degree*** of a vertex is the number of edges leaving that vertex.
- Nodes with out-degree=0 (no children) are called the ***leaves*** of the tree.
- ***Root*** is the only node in the tree with in-degree=0.
- The ***child*** *C* of a node *A* is the node that is connected to *A* by a single edge. Then, *A* is the ***parent*** of *C*.
- ***Grandparent*** and ***grandchild*** relations can be defined in the same manner.
- Nodes with the same parent are called ***siblings***.

More definitions

- A ***path*** from a node n_1 to n_k is defined as a sequence of nodes $n_1, \ldots, n_k$ such that n_i is the parent of n_{i+1} for $1 \le i < k$.
  - The ***length*** of this path is the number of edges on the path, i.e., $k-1$.
  - There is **exactly** one path from the root to each node.
- The ***depth*** of any node n_i is the *length of the unique path from the root to* n_i.
  - The depth of root is 0.
- The ***height*** of any node n_i is the *length of the longest path from* n_i *to a leaf*.

A General Tree Example

- *Root*
  - **A**
- *Children* of A
  - **B**, **C**, **D**, **E**, **F**.
- Leaves of the tree
  - **C**, **D**, **G**, **H**, **I**, **K**, **L**, **M**, and **N**.
- Siblings of **G**
  - **H and I**.



Remarks

The ***height*** of all *leaves* are 0.

The ***height*** of a *tree* is the *height of its root*.

The height of the above tree is 3.

The height of C is 0.

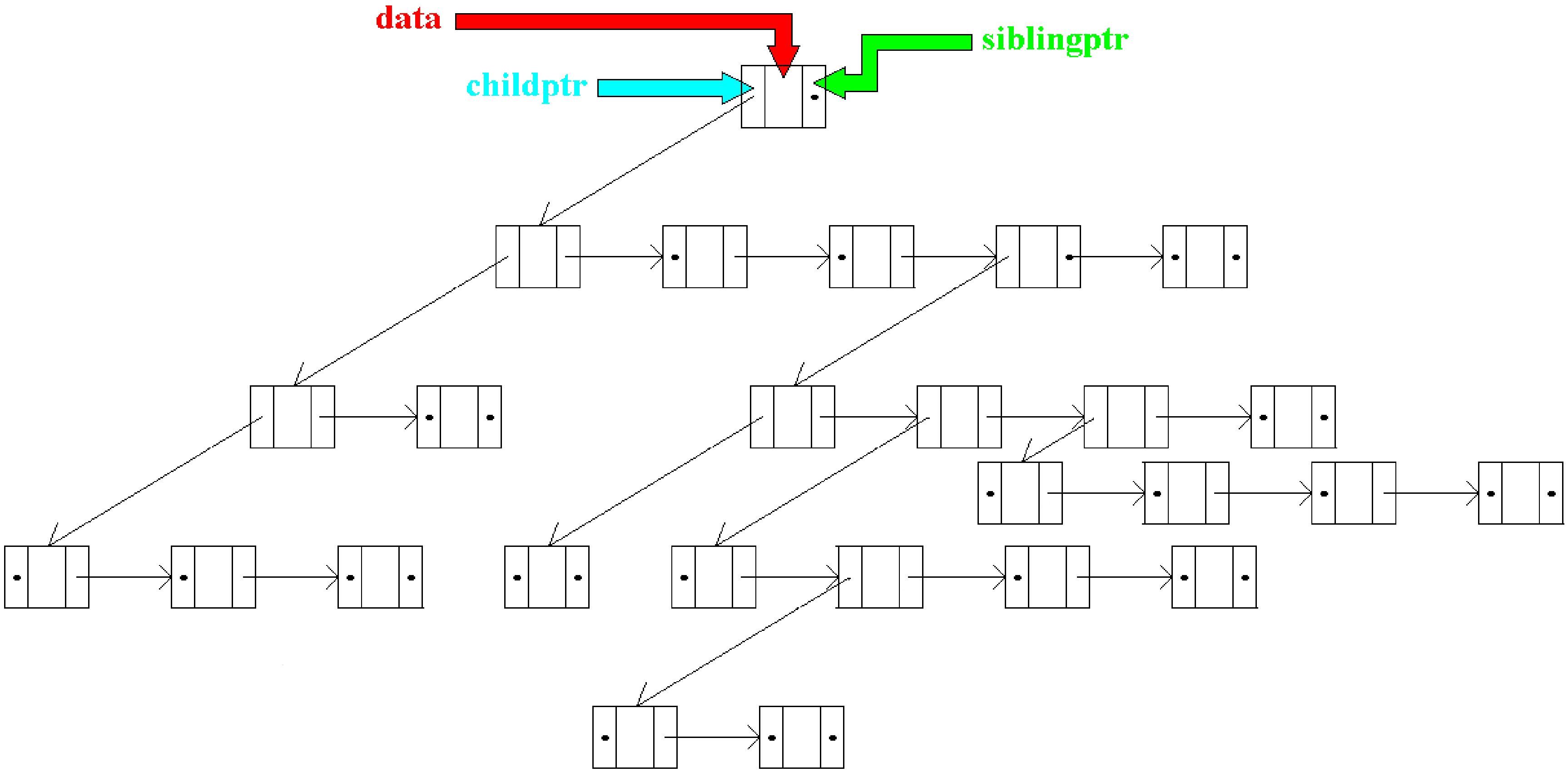
If there is a ***path*** from n_1 to n_2, then n_1 is an *ancestor* of n_2 and n_2 is a *descendant* of n_1 .

Implementation of Trees

- One way to implement a tree would be to have a pointer in each node to each child, besides the data it holds. This is infeasible!
- Q:Why?
- A:The number of children of each node is variable, and hence, unknown.
- Another option would be to keep the children in a linked list where the first node of the list is pointed to by the pointer in the parent node as a header.
- A typical tree node declaration in C would be as follows:

```
struct TNode_type {
      int data;
      struct TNodetype *childptr, *siblingptr;
}
```

A General Tree Implementation



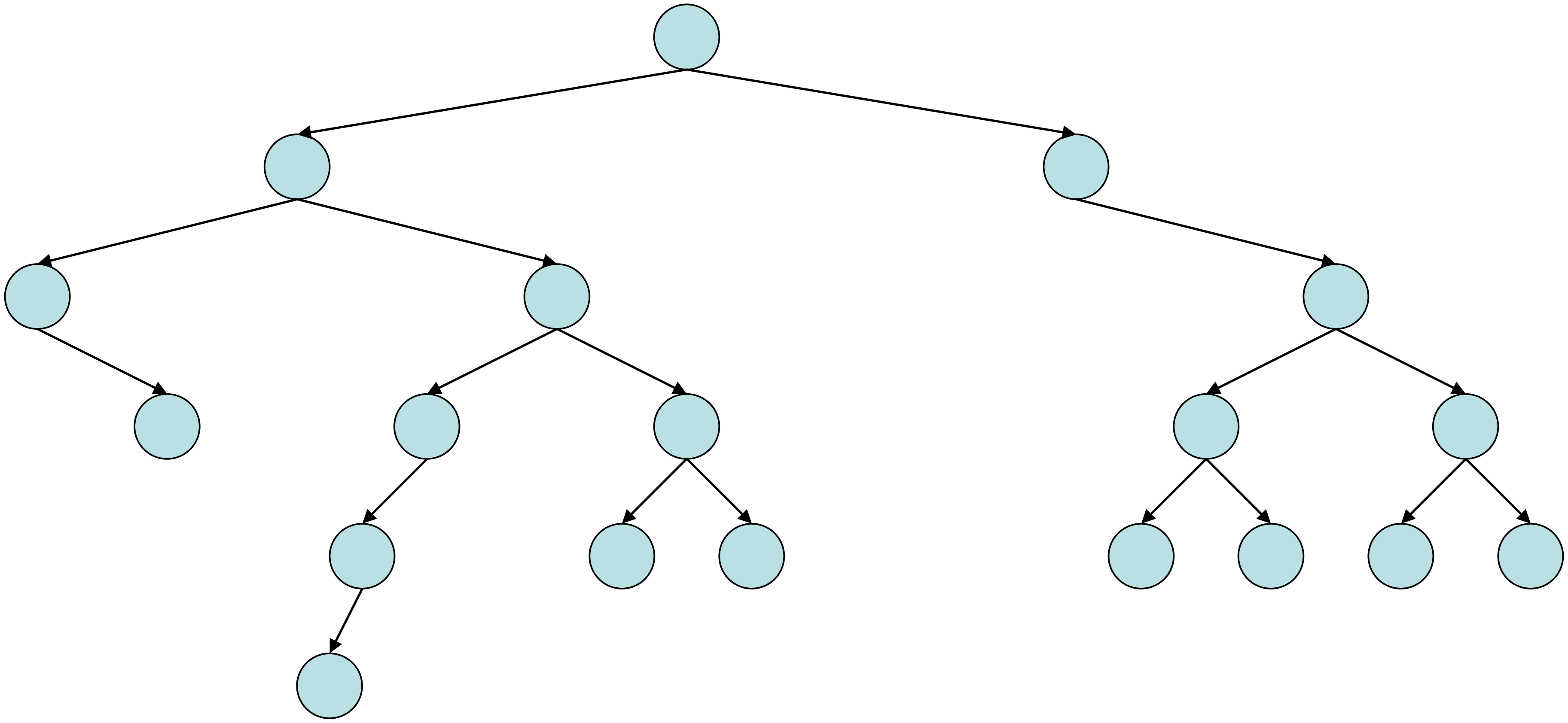
Binary Trees (BTs)

- A *binary tree* is a tree of nodes *with at most two children*.
- Declaration of a typical node in a binary tree
  - struct BTNodeType {

```
struct BTNodeType {
    infoType *data;
    struct BTNodeType *left;
    struct BTNodeType *right;
}
```

- Average depth in BTs: $O(\sqrt{n})$

A Binary Tree Example



Tree Traversals

- A tree can be traversed recursively in three different ways:
  - in-order traversal (left-node-right or LNR)
    - First recursively visit the left subtree.
    - Next process the data in the current node.
    - Finally, recursively visit the right subtree.
  - pre-order traversal (node-left-right or NLR)
  - post-order traversal (left-right-node or LRN)

Recursive In-order (LNR) Traversal

```
void in-order(BTNodeType *p) {
  if (p!=null){
      in-order(p->left);
      operate(p);
      in-order(p->right);
  }
}
```

Recursive Pre-order (NLR) Traversal

```
void pre-order(BTNodeType *p) {
  if (p!=null){
      operate(p);
      pre-order(p->left);
      pre-order(p->right);
  }
}
```

Recursive Post-order (LRN) Traversal

```
void post-order(BTNodeType *p) {
  if (p!=null){
      post-order(p->left);
      post-order(p->right);
      operate(p);
  }
}
```

Expression Trees

- Prefix, postfix and infix formats of arithmetic expressions

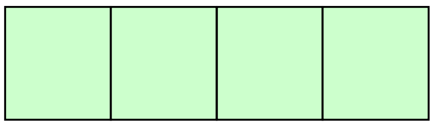
Infix	Postfix	Prefix
A+B	AB+	+AB
A/(B+C)	ABC+/	/A+BC
A/B+C	AB/C+	+/ABC
A-B*C+D/(E+F)	ABC*-DEF+/+	+-A*BC/D+EF
A*((B+C)/(D-E)+F)-G/(H-I)	ABC+DE-/F+*GHI-/-	-*A+/+BC-DEF/G-HI

Construction of an Expression Tree from Postfix Expressions

- Initialize an empty ***stack of subtrees***
- Repeat
  - Get token;
  - if token is an operand
    - Push it as a subtree
  - else
    - pop the last two subtrees and form & push a subtree such that root is the current operator and left and right operands are the former and latter subtrees, respectively
- Until the end of arithmetic expression

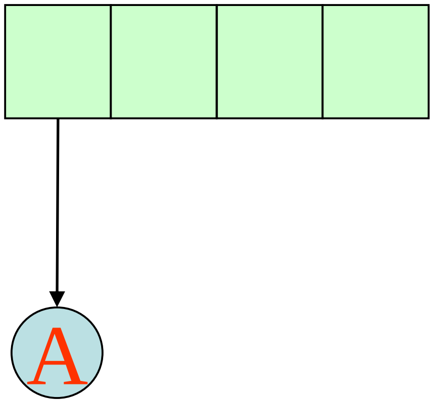
Example to Construction of Expression Trees

Empty Stack of subtrees



ABC+DE-/F+*GHI-/-

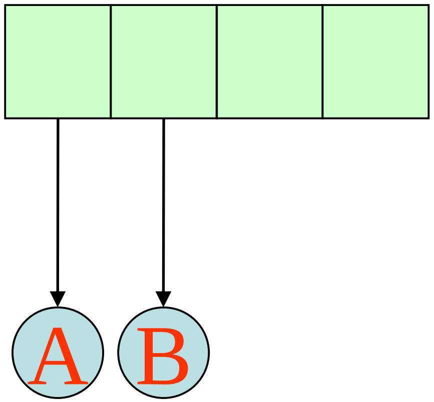
Example to Construction of Expression Trees - 1



Stack of subtrees

ABC+DE-/F+*GHI-/-

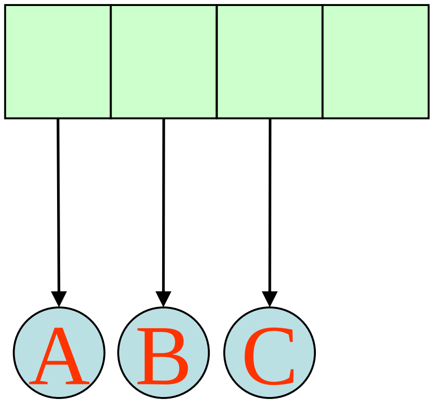
Example to Construction of Expression Trees - 2



Stack of subtrees

ABC+DE-/F+*GHI-/-

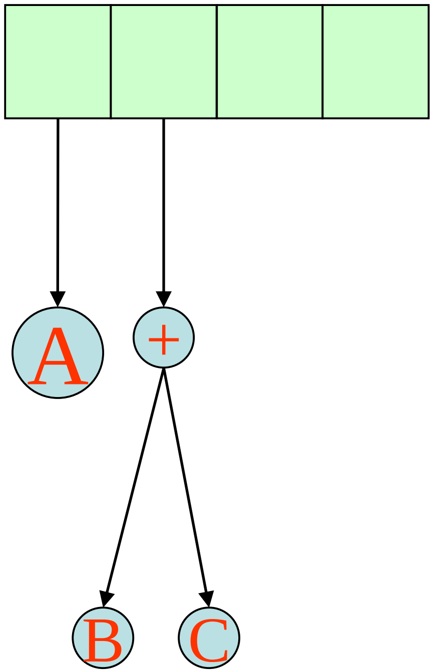
Example to Construction of Expression Trees - 3



Stack of subtrees

ABC+DE-/F+*GHI-/-

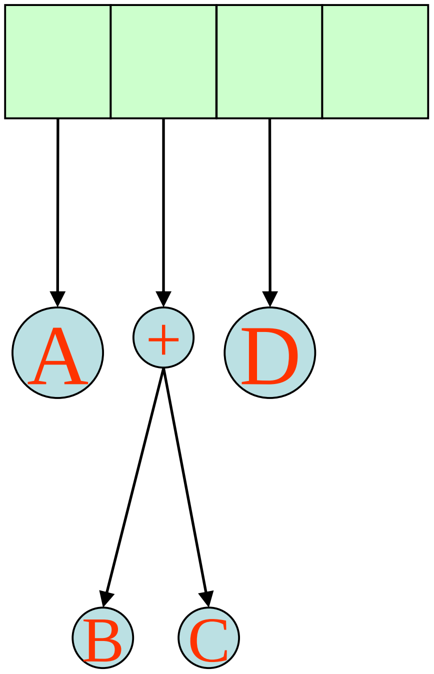
Example to Construction of Expression Trees - 4



Stack of subtrees

ABC+DE-/F+*GHI-/-

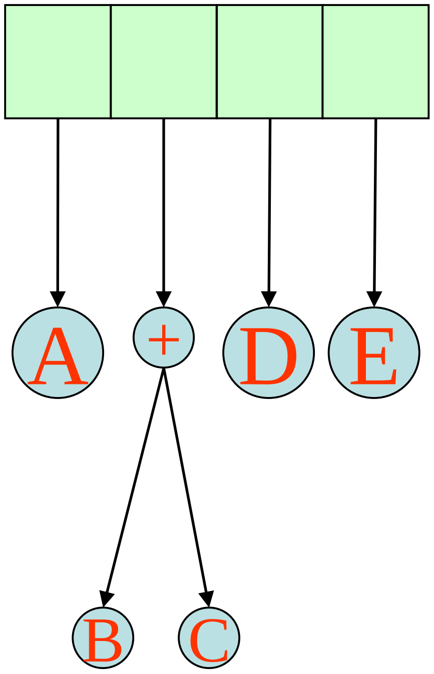
Example to Construction of Expression Trees - 5



Stack of subtrees

ABC+DE-/F+*GHI-/-

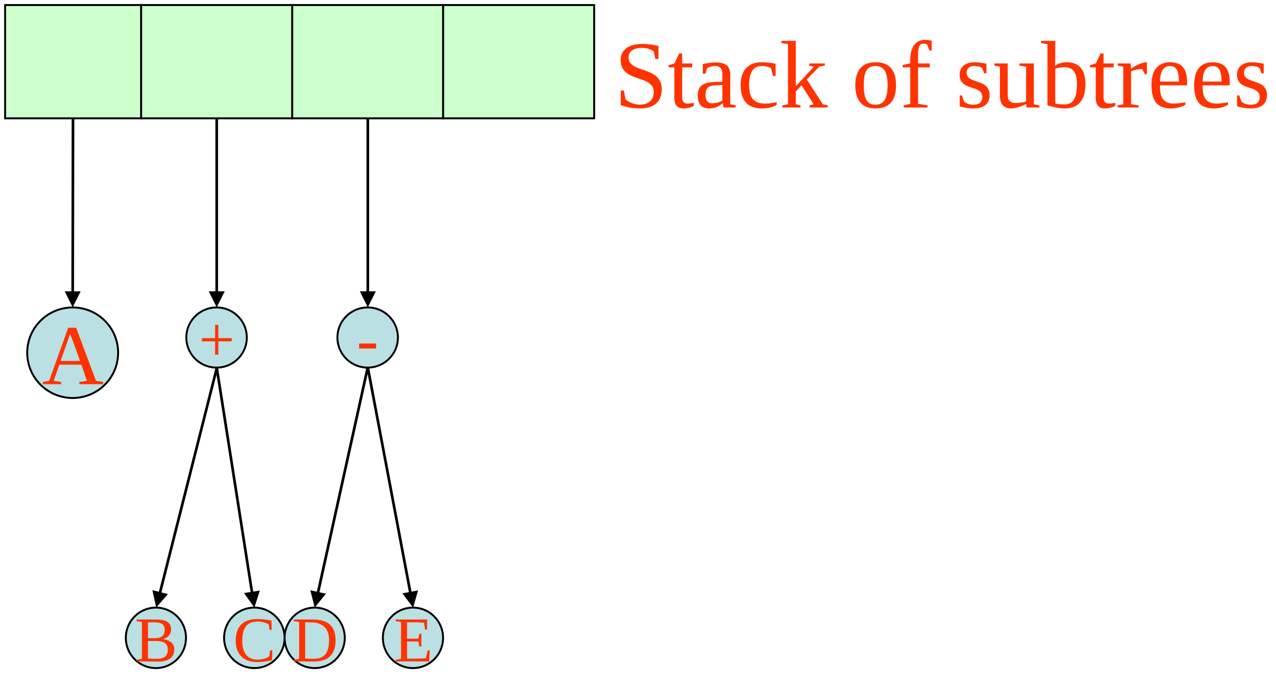
Example to Construction of Expression Trees - 6



Stack of subtrees

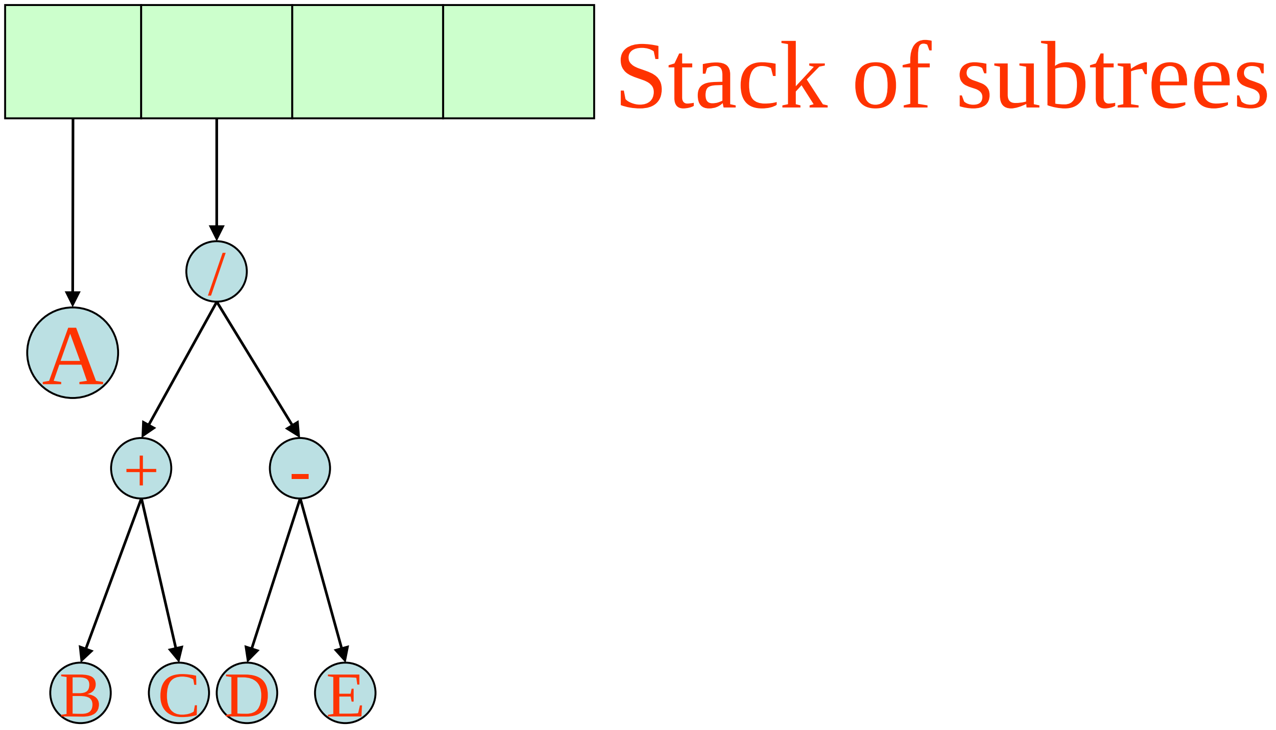
ABC+DE-/F+*GHI-/-

Example to Construction of Expression Trees - 7



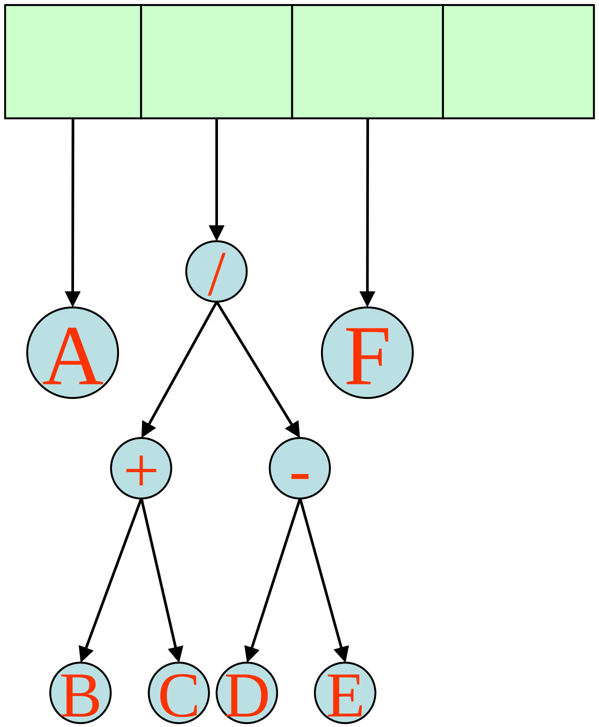
ABC+DE-/F+*GHI-/-

Example to Construction of Expression Trees - 8



ABC+DE-/F+*GHI-/-

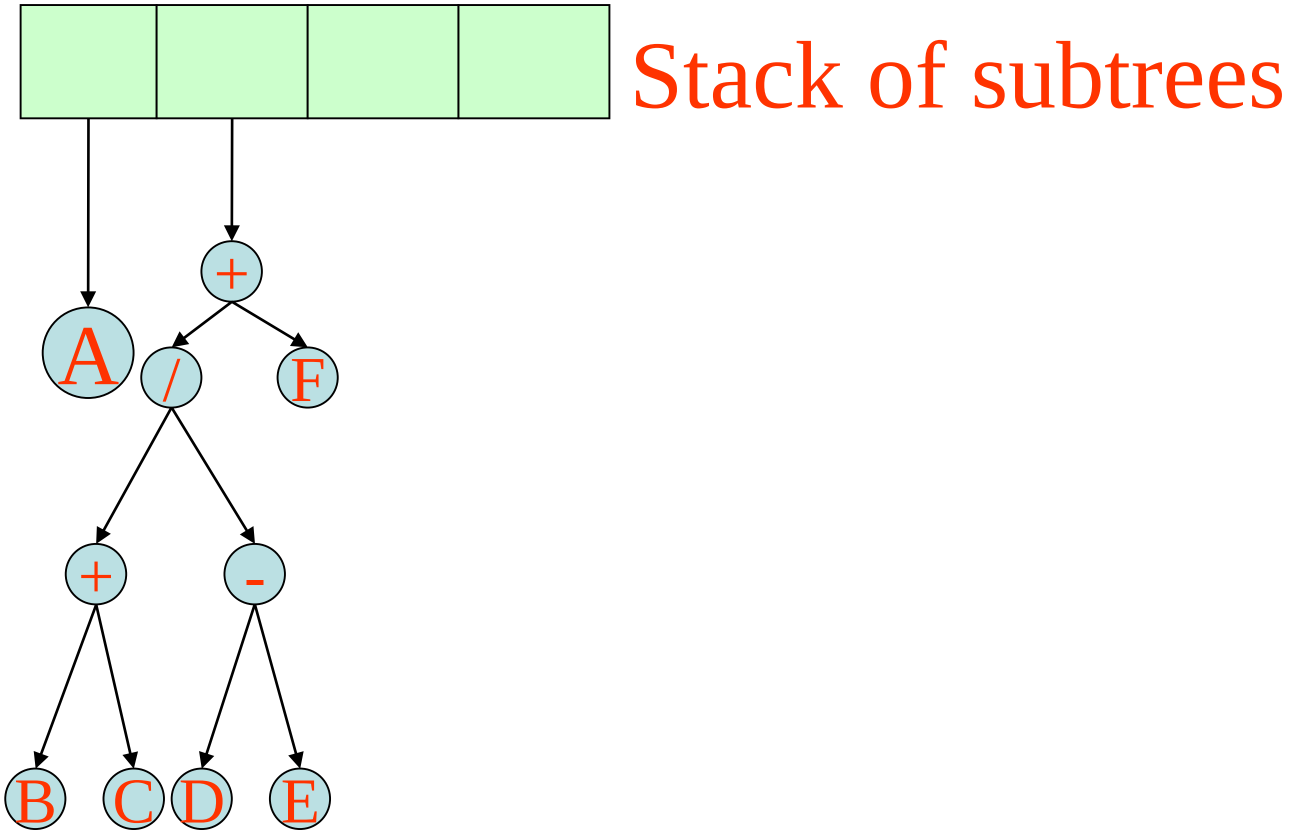
Example to Construction of Expression Trees - 9



Stack of subtrees

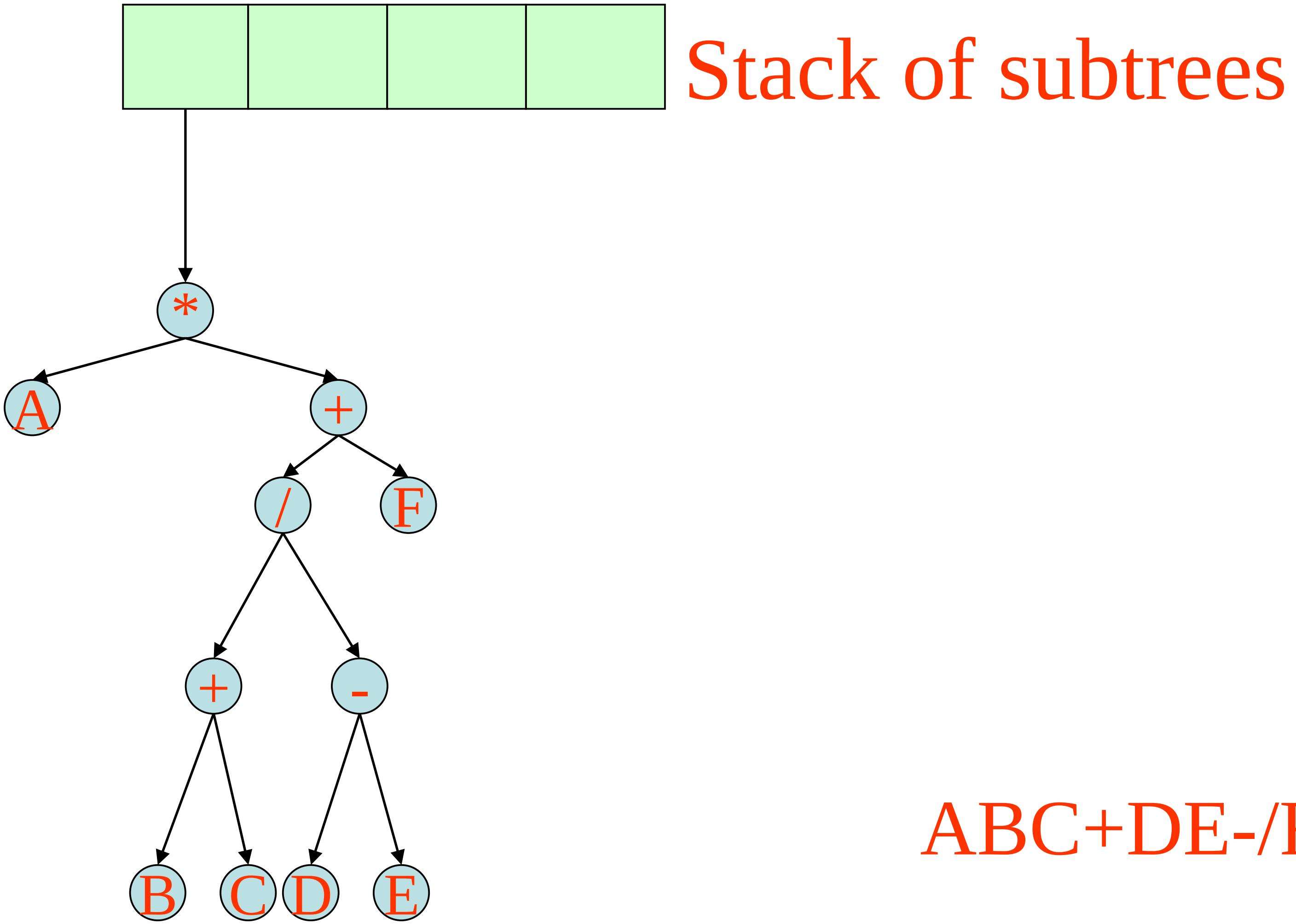
ABC+DE-/F+*GHI-/-

Example to Construction of Expression Trees - 10



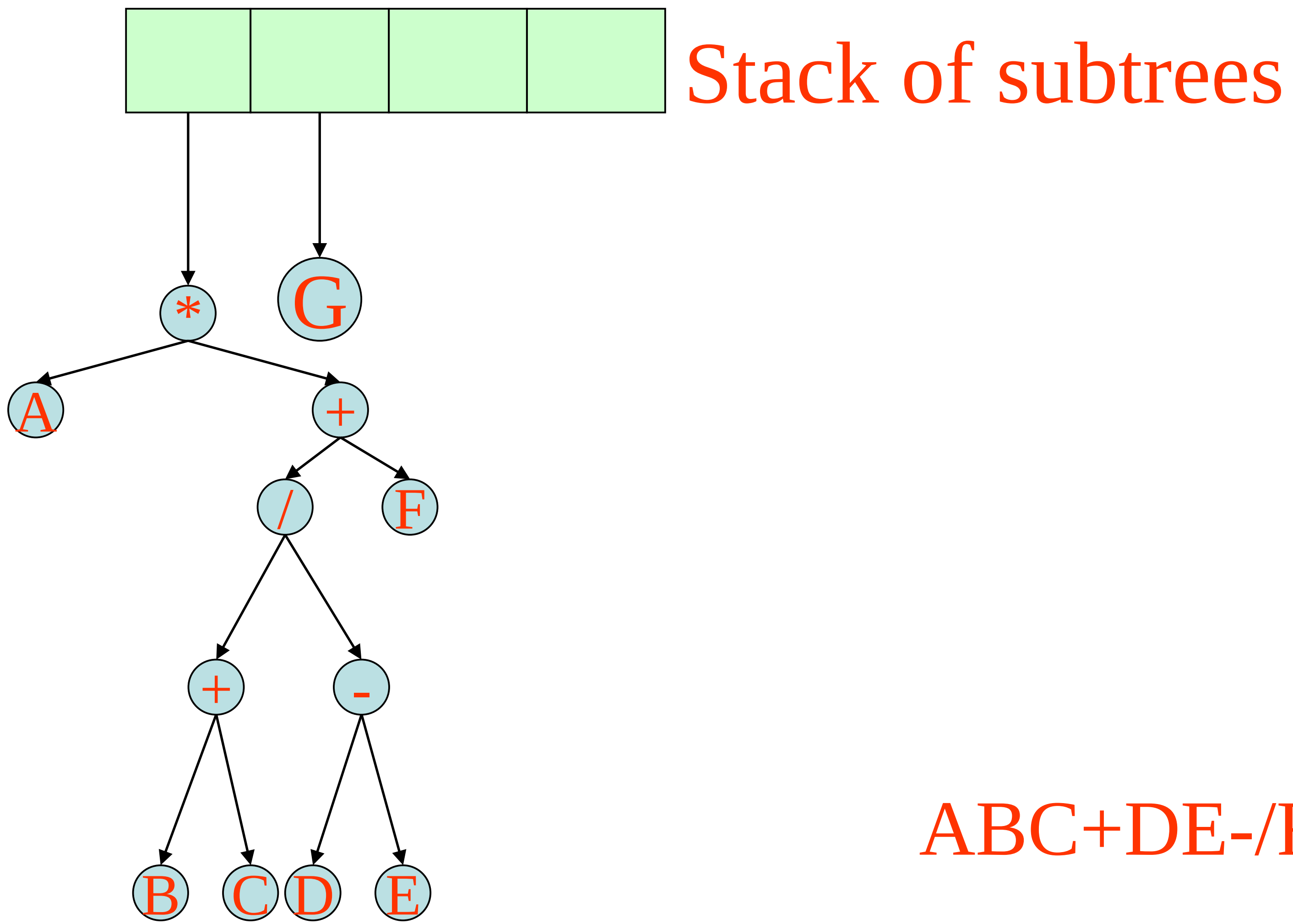
ABC+DE-/F+*GHI-/-

Example to Construction of Expression Trees - 11



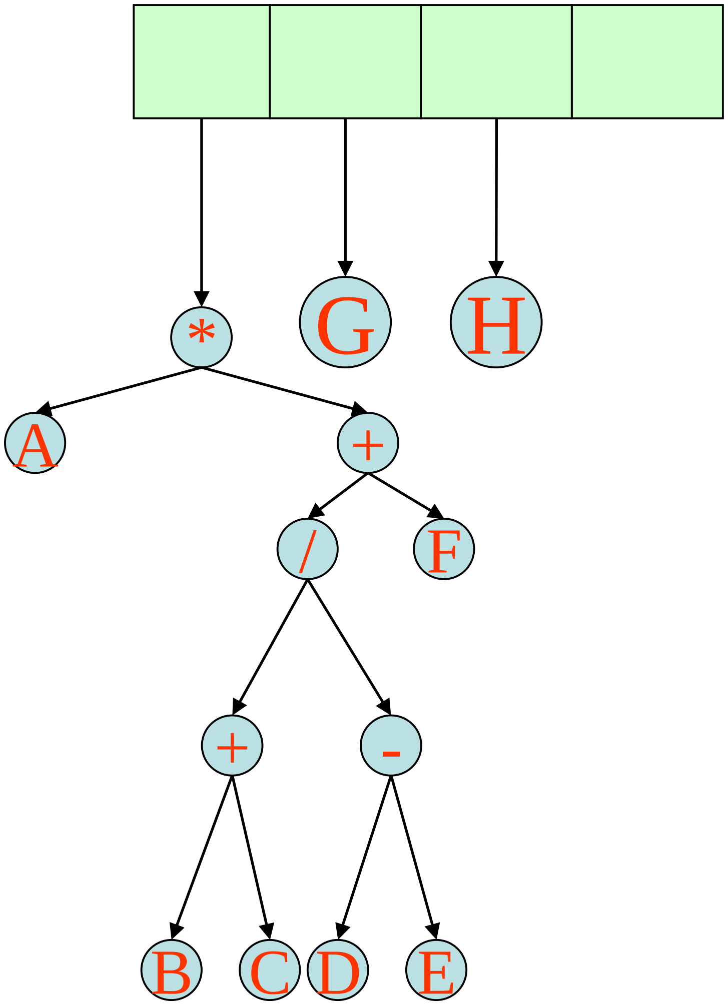
ABC+DE-/F+*GHI-/-

Example to Construction of Expression Trees - 12



ABC+DE-/F+*GHI-/-

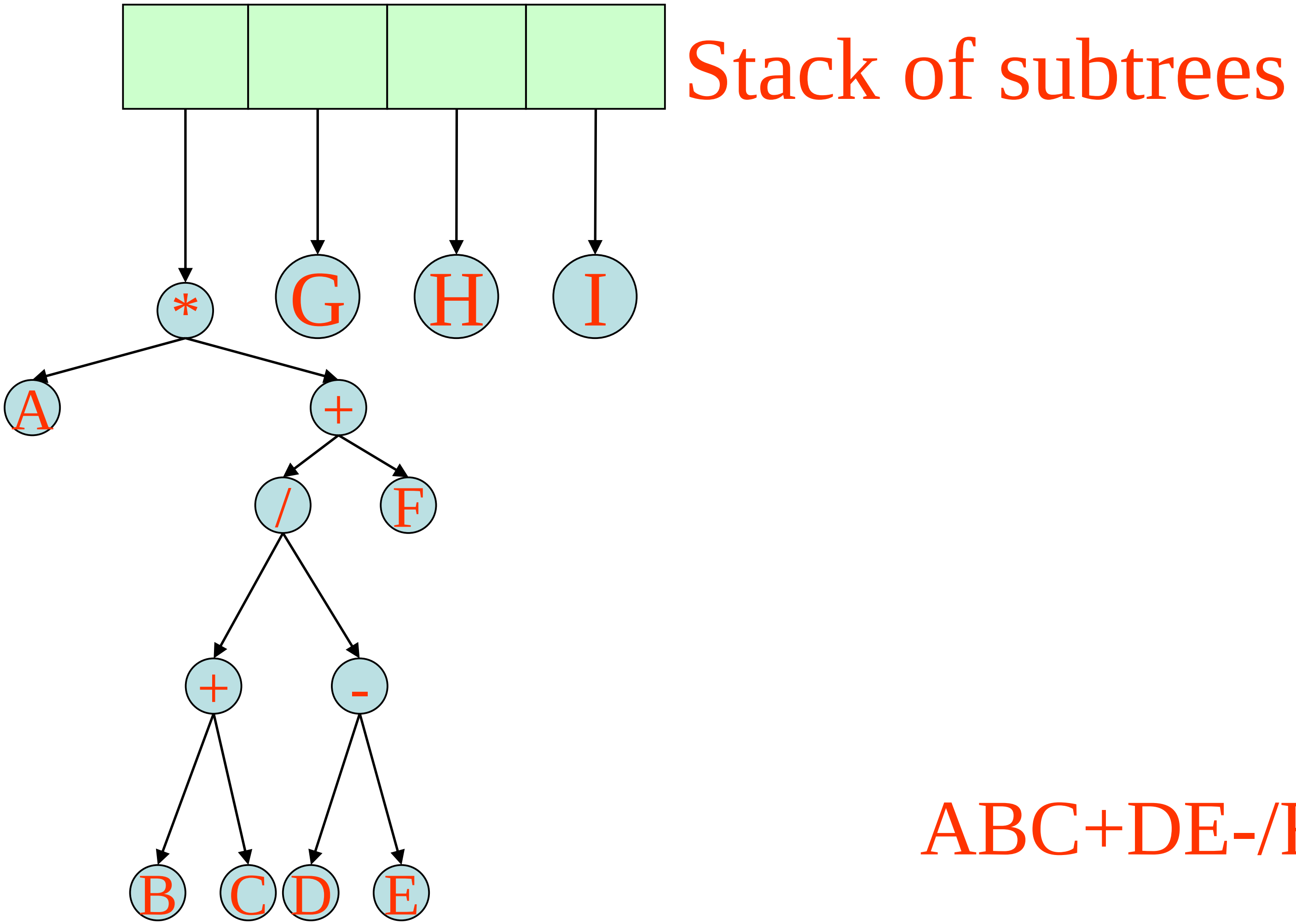
Example to Construction of Expression Trees - 13



Stack of subtrees

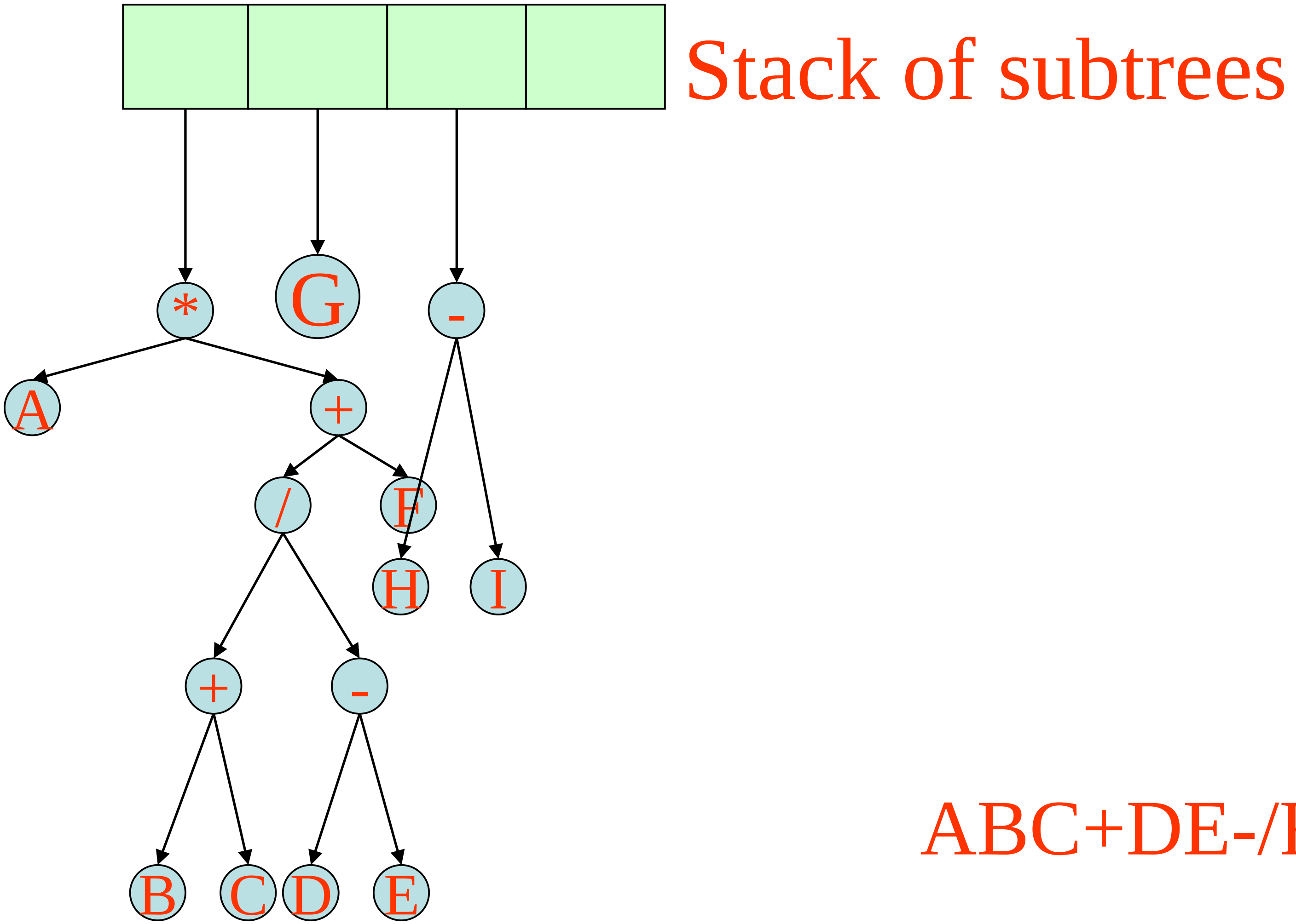
ABC+DE-/F+*GHI-/-

Example to Construction of Expression Trees - 14



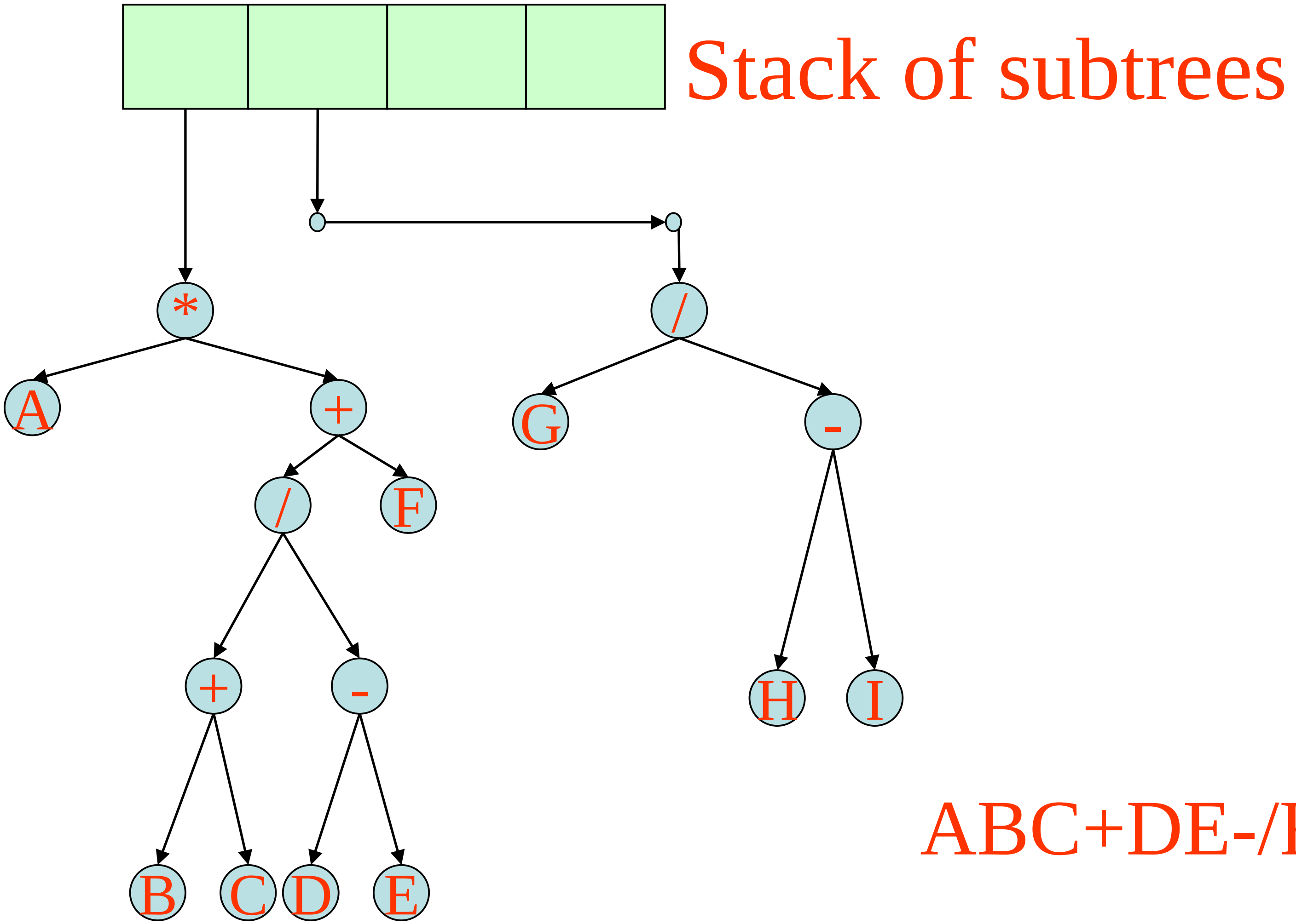
ABC+DE-/F+*GHI-/-

Example to Construction of Expression Trees - 15



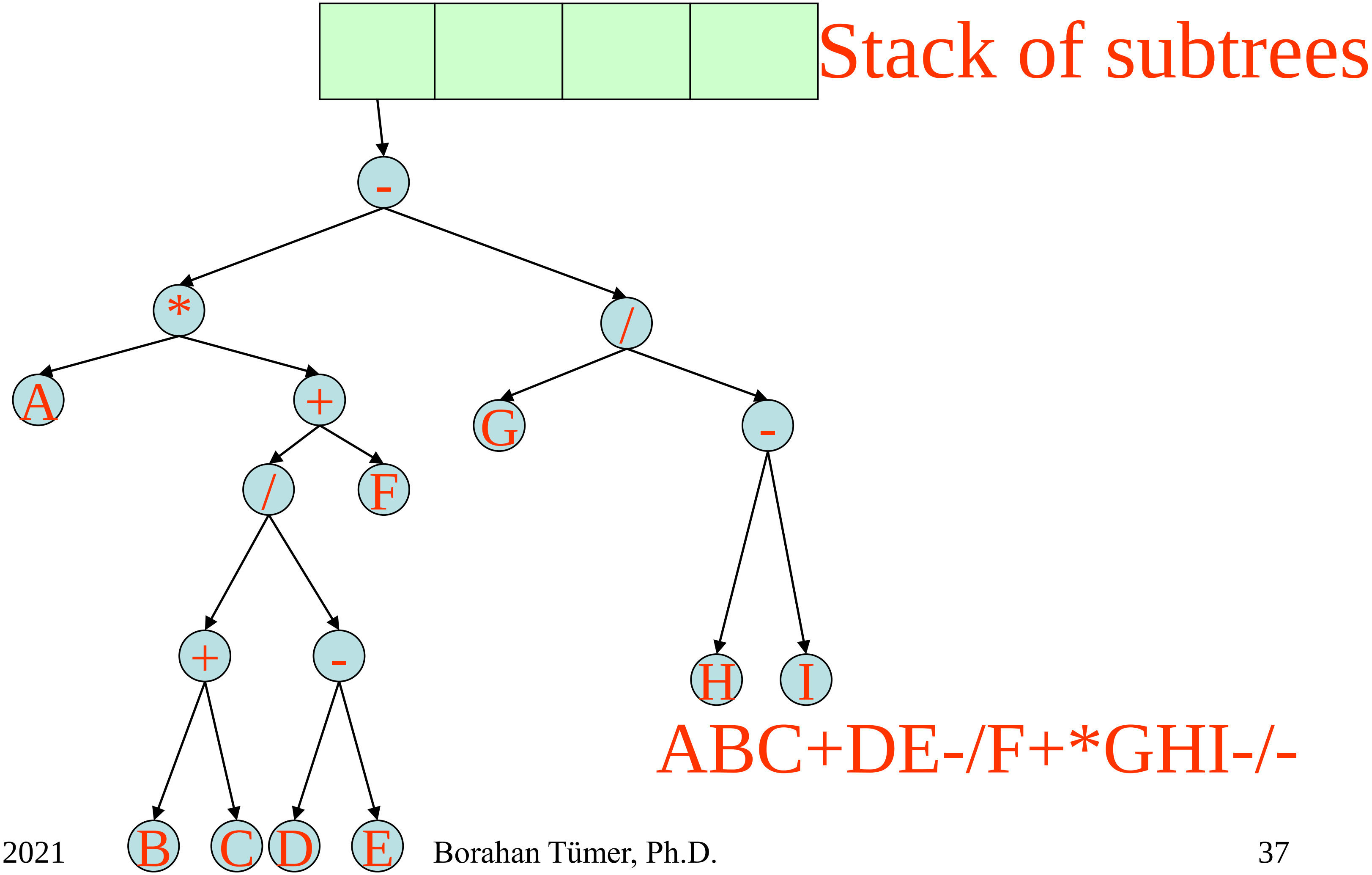
ABC+DE-/F+*GHI-/-

Example to Construction of Expression Trees - 16



ABC+DE-/F+*GHI-/-

Example to Construction of Expression Trees - 17



ABC+DE-/F+*GHI-/-

Binary Search Trees (BSTs)

BSTs are binary trees with keys placed in each node in such a manner that

the key of a node is

greater than all keys in its left sub-tree

and

***less than all keys in its right sub-tree*.**

Here, we assume *no replication of keys*. For replicating keys, the relations are modified as "*greater than or equal to*" or "*less than or equal to*" depending on the application.

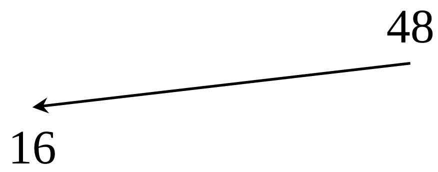
A Binary Search Tree Example

48	16	24	20	32	8	12	54	72	18	96	64	17	60	98	68	84	36	30

48

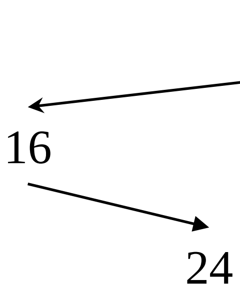
A Binary Search Tree Example

48	16	24	20	32	8	12	54	72	18	96	64	17	60	98	68	84	36	30



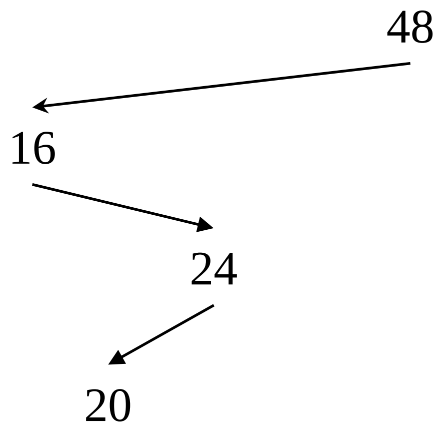
A Binary Search Tree Example

48	16	24	20	32	8	12	54	72	18	96	64	17	60	98	68	84	36	30



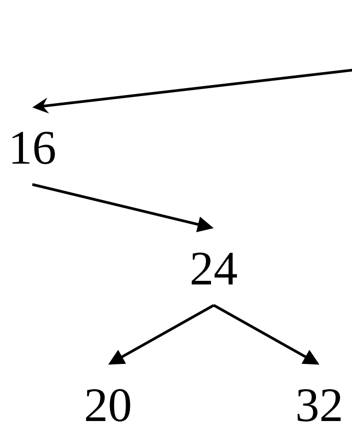
A Binary Search Tree Example

48	16	24	20	32	8	12	54	72	18	96	64	17	60	98	68	84	36	30



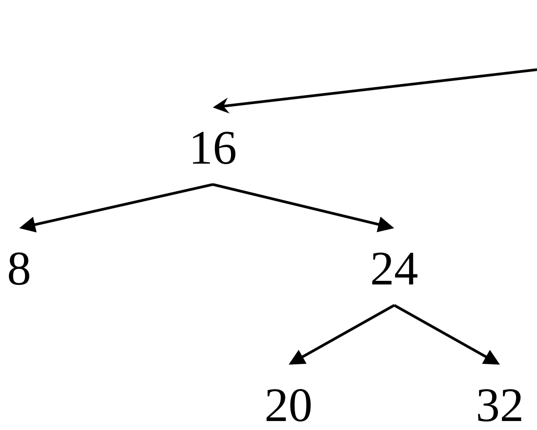
A Binary Search Tree Example

48	16	24	20	32	8	12	54	72	18	96	64	17	60	98	68	84	36	30



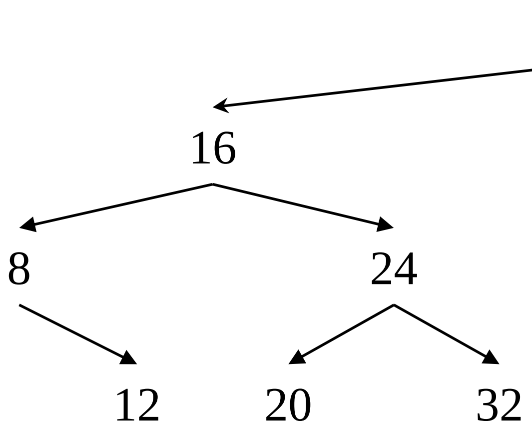
A Binary Search Tree Example

48	16	24	20	32	8	12	54	72	18	96	64	17	60	98	68	84	36	30



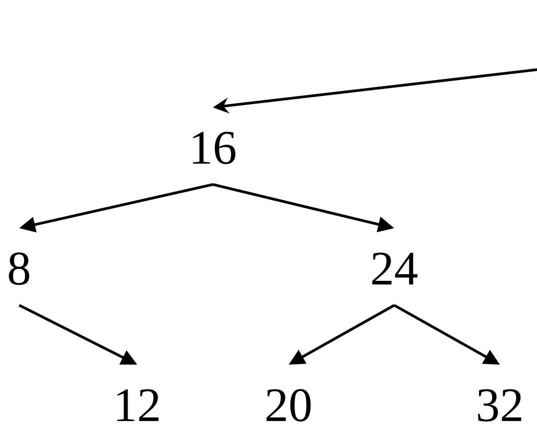
A Binary Search Tree Example

48	16	24	20	32	8	12	54	72	18	96	64	17	60	98	68	84	36	30



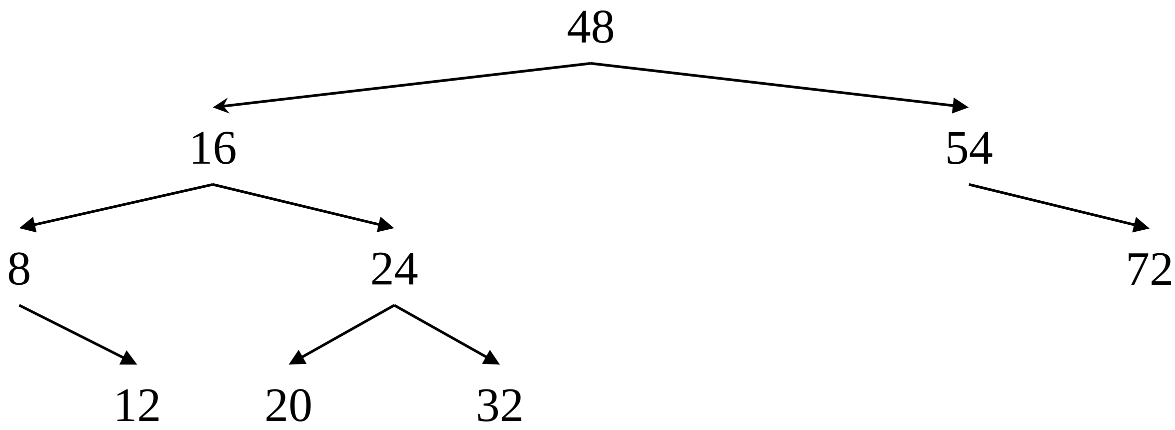
A Binary Search Tree Example

48	16	24	20	32	8	12	54	72	18	96	64	17	60	98	68	84	36	30



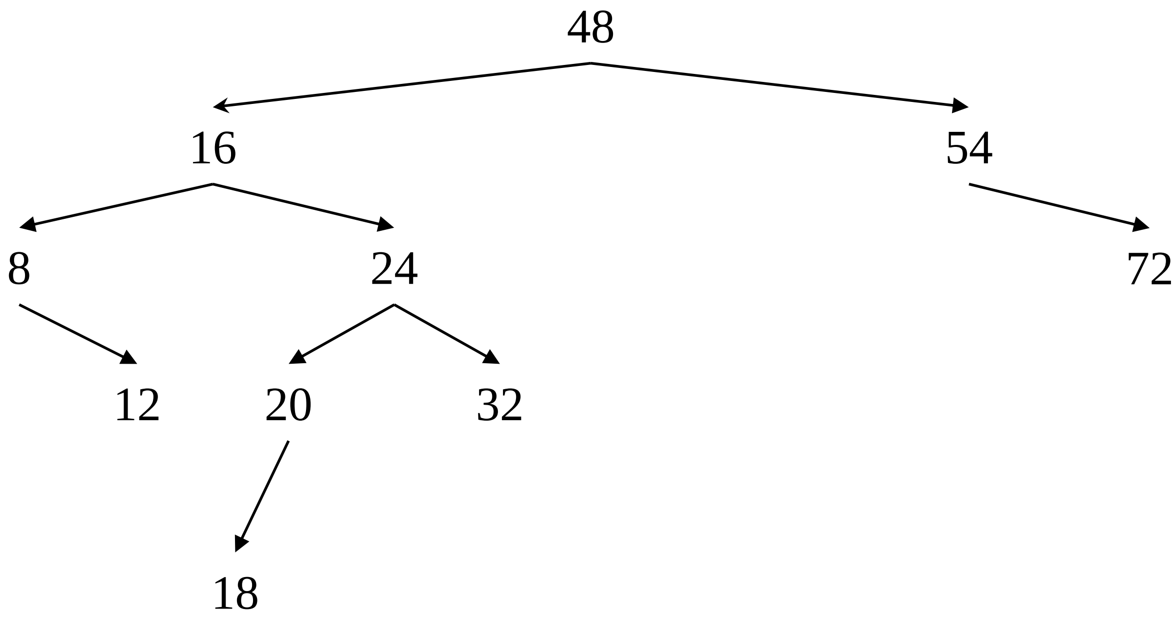
A Binary Search Tree Example

48	16	24	20	32	8	12	54	72	18	96	64	17	60	98	68	84	36	30



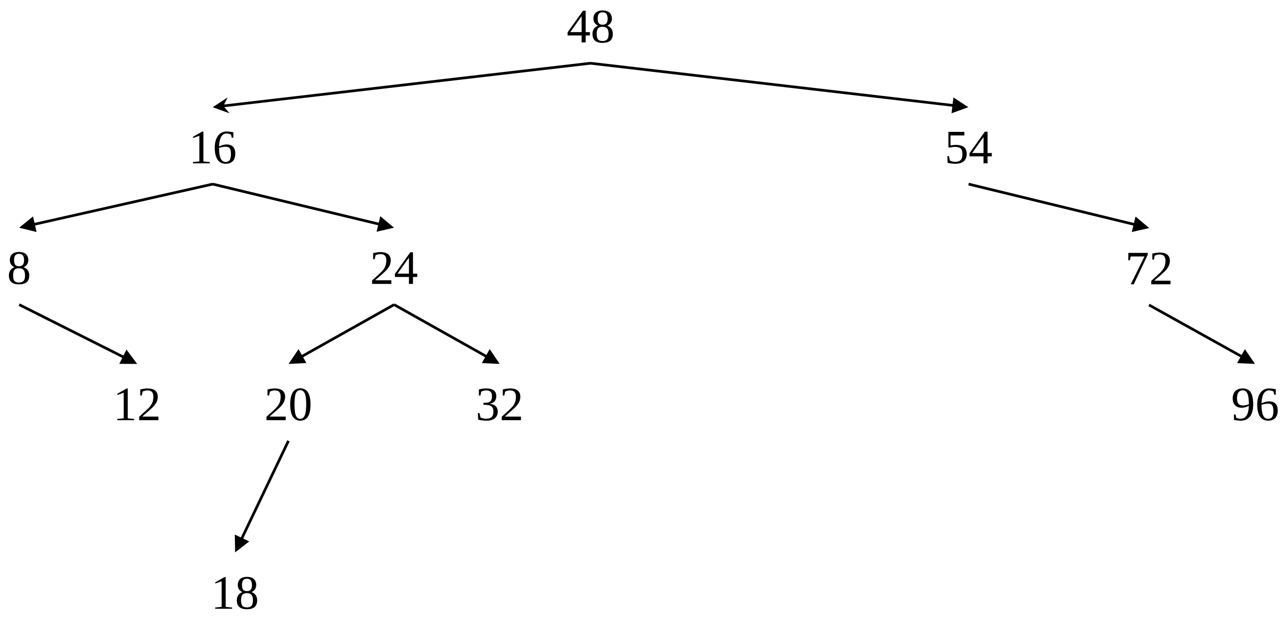
A Binary Search Tree Example

48	16	24	20	32	8	12	54	72	18	96	64	17	60	98	68	84	36	30



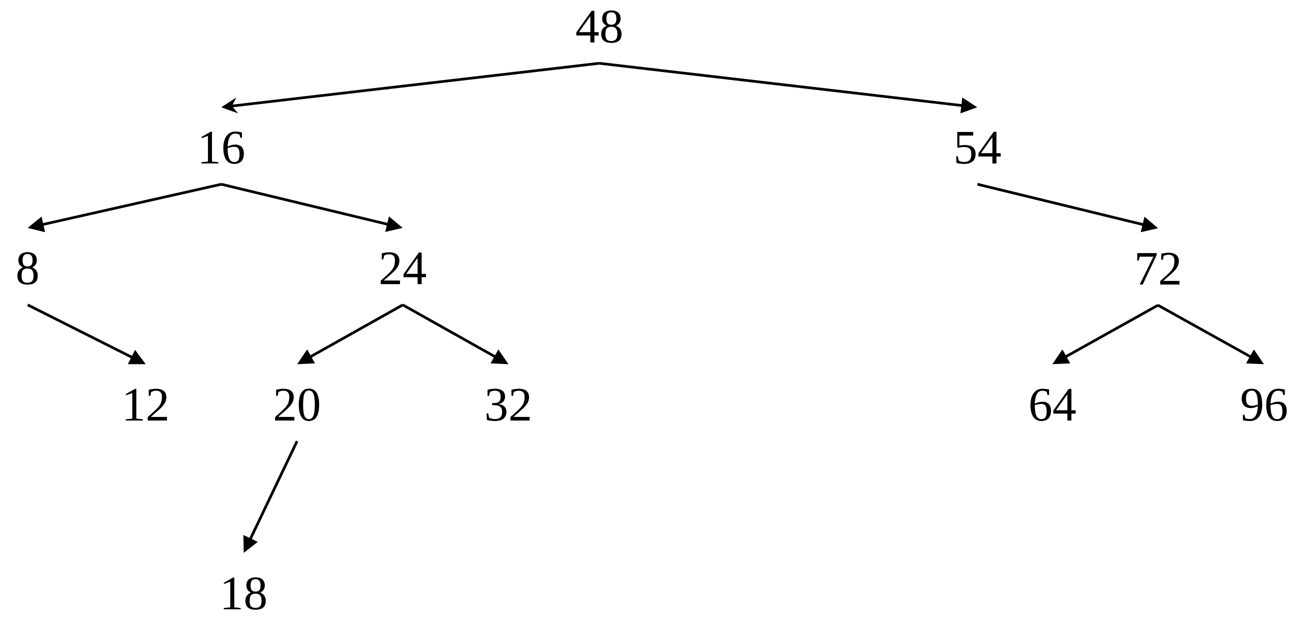
A Binary Search Tree Example

48	16	24	20	32	8	12	54	72	18	96	64	17	60	98	68	84	36	30



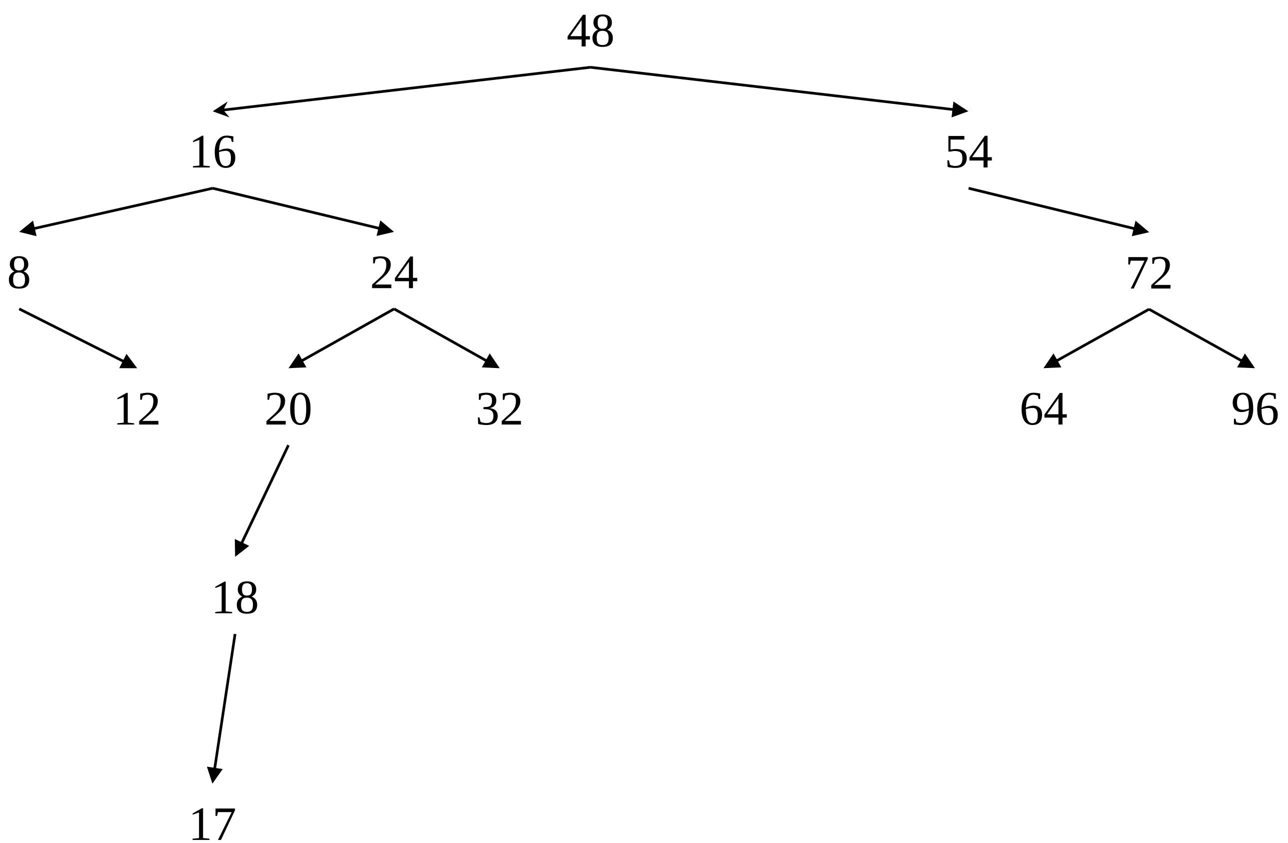
A Binary Search Tree Example

48	16	24	20	32	8	12	54	72	18	96	64	17	60	98	68	84	36	30



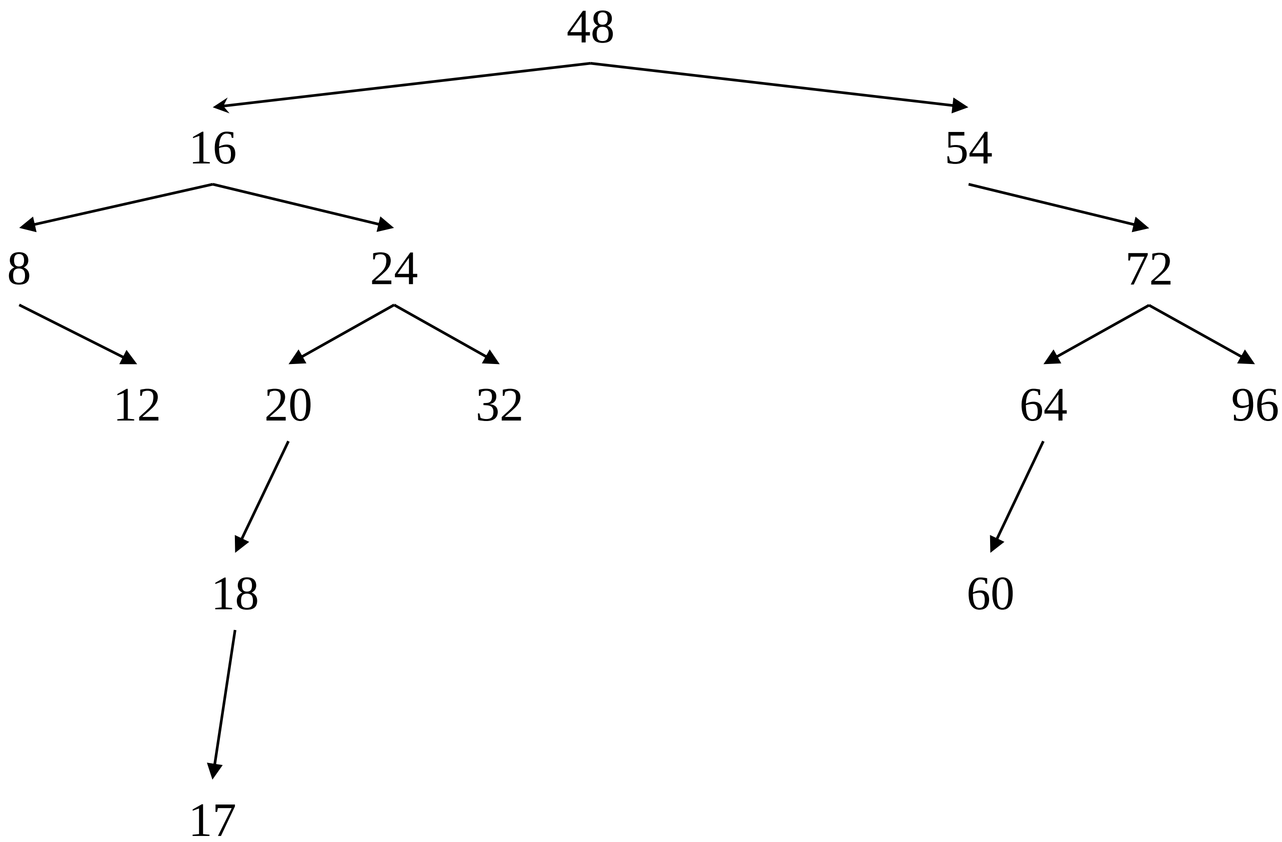
A Binary Search Tree Example

48	16	24	20	32	8	12	54	72	18	96	64	17	60	98	68	84	36	30



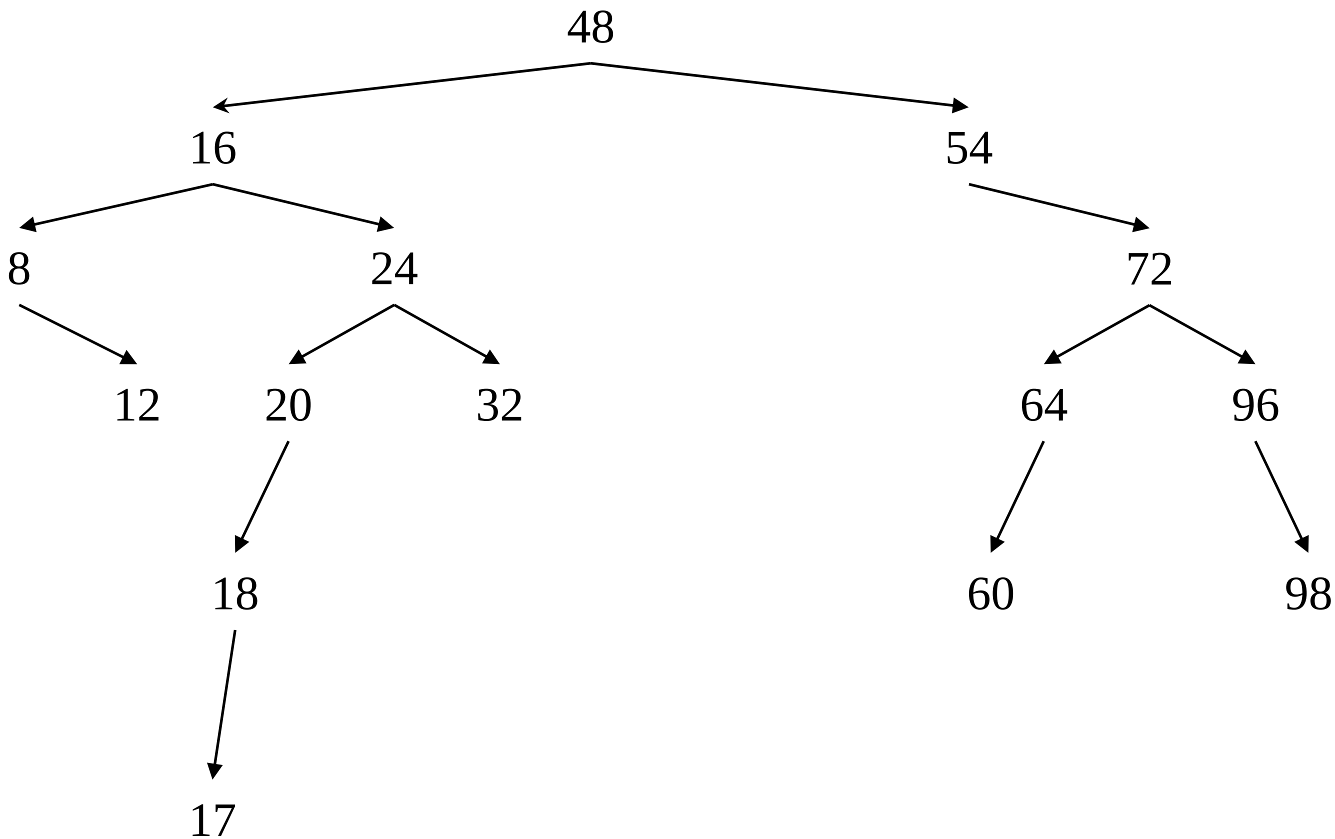
A Binary Search Tree Example

48	16	24	20	32	8	12	54	72	18	96	64	17	60	98	68	84	36	30



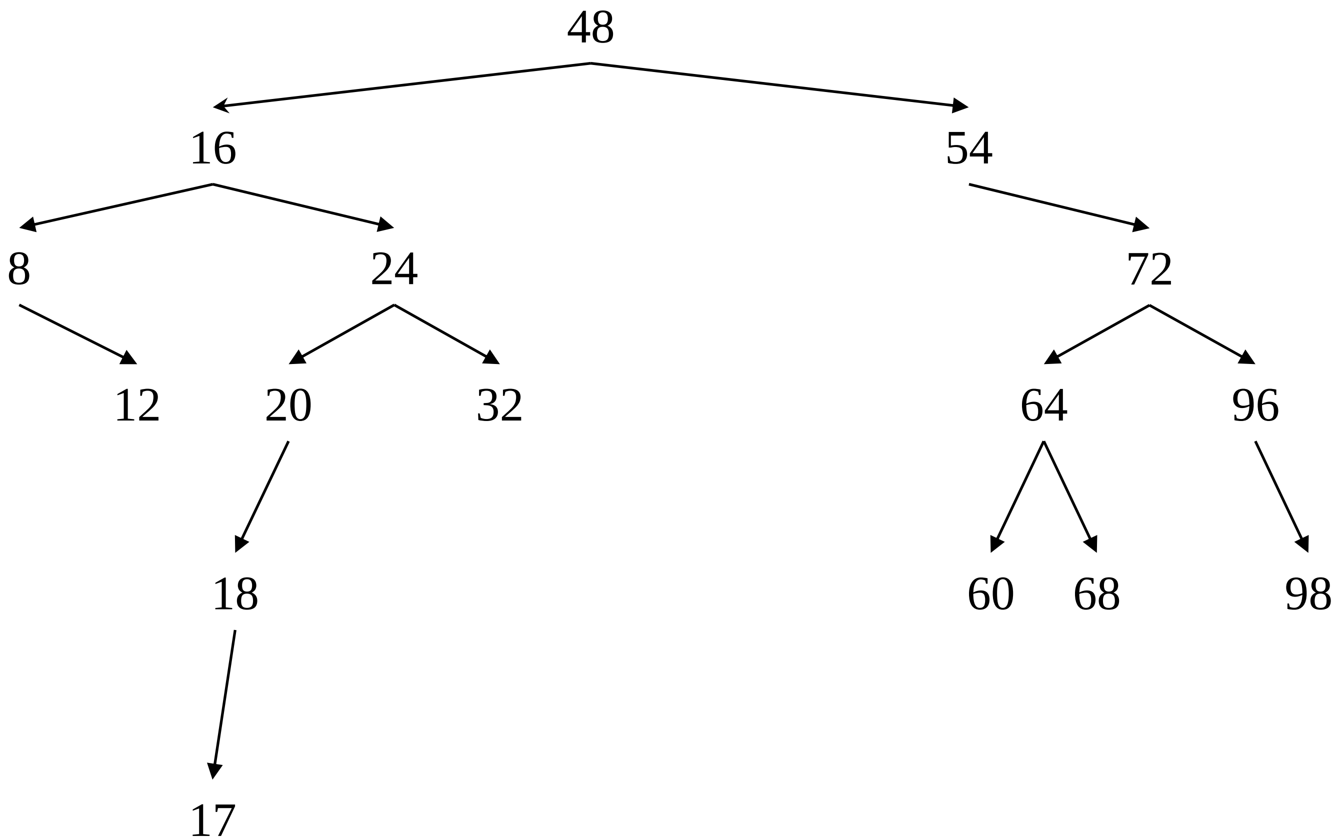
A Binary Search Tree Example

48	16	24	20	32	8	12	54	72	18	96	64	17	60	98	68	84	36	30



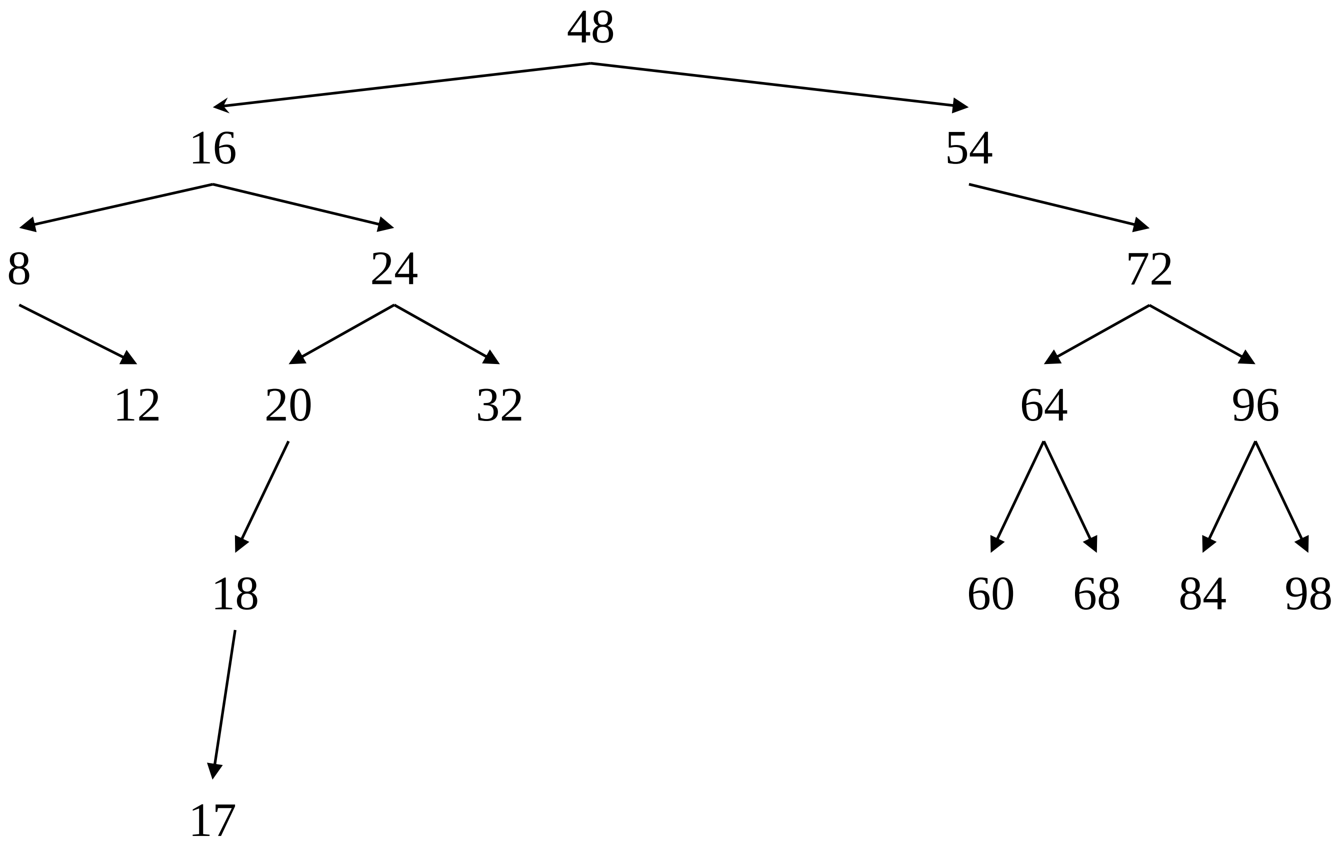
A Binary Search Tree Example

48	16	24	20	32	8	12	54	72	18	96	64	17	60	98	68	84	36	30



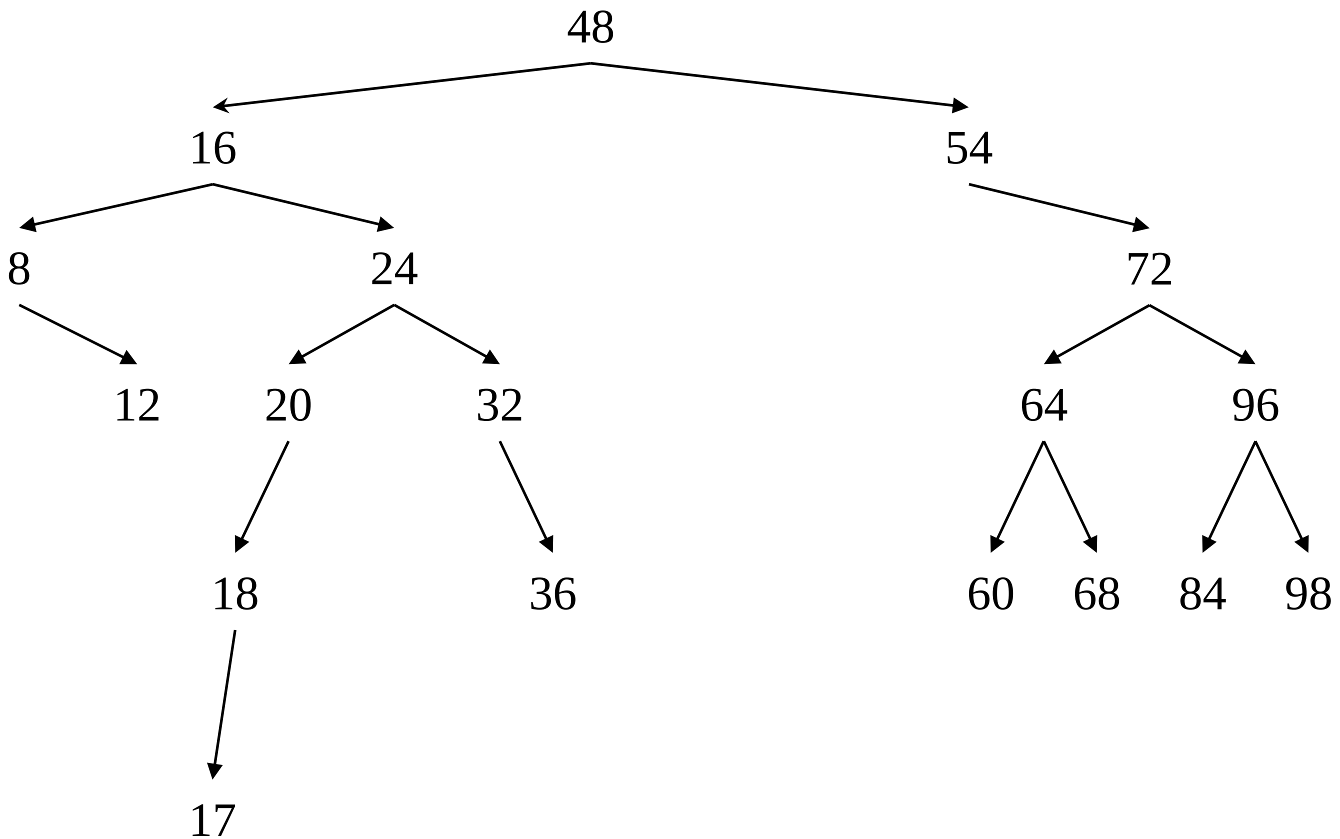
A Binary Search Tree Example

48	16	24	20	32	8	12	54	72	18	96	64	17	60	98	68	84	36	30



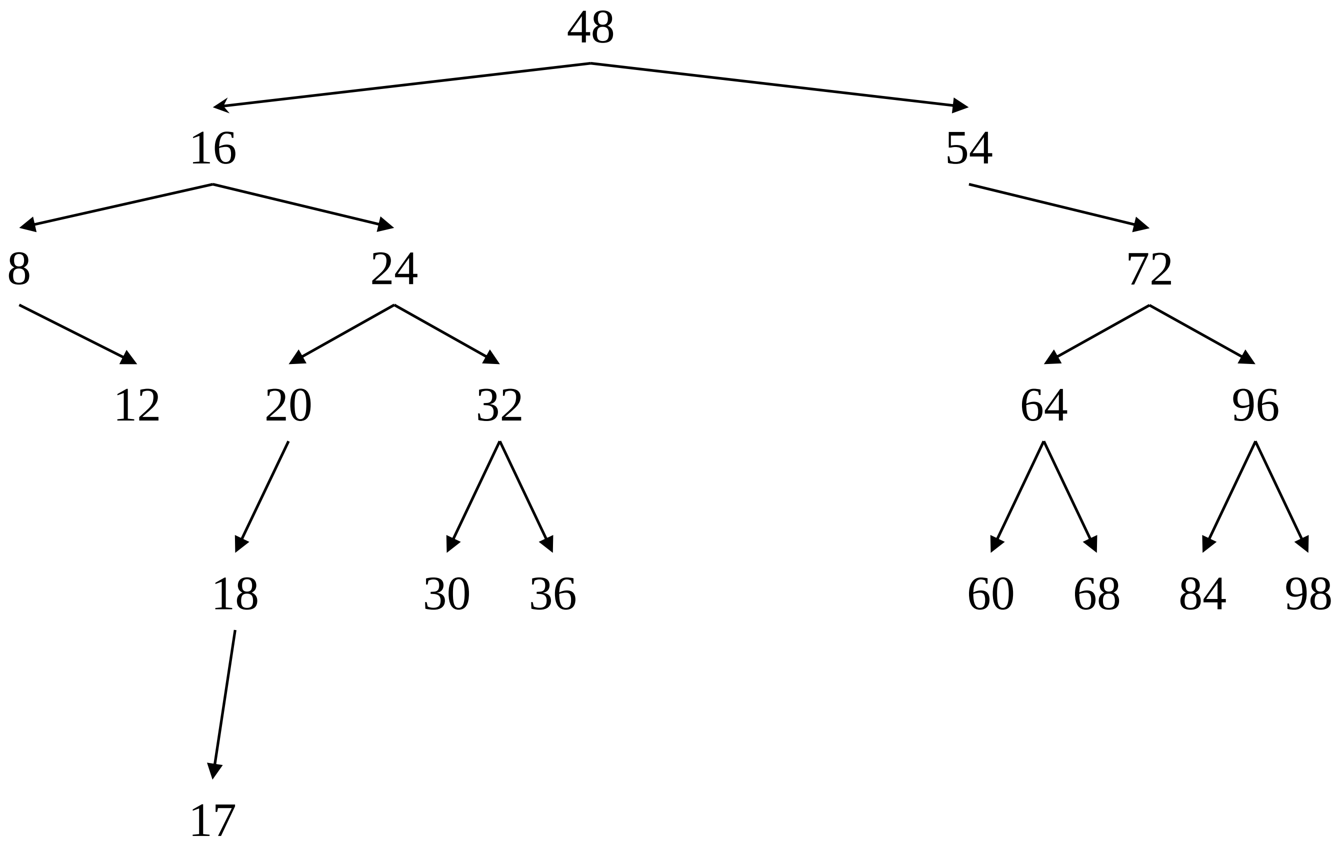
A Binary Search Tree Example

48	16	24	20	32	8	12	54	72	18	96	64	17	60	98	68	84	36	30



A Binary Search Tree Example

48	16	24	20	32	8	12	54	72	18	96	64	17	60	98	68	84	36	30



Recursive Find Function in BSTs

```
int find (BTNodeType *p, int key)
{ // we assume infotype in BTNodeType is int
   if (p != NULL)
      if (key < p->data) find(p->left, key);
      else if (key > p->data) find(p->right, key);
            else if (p->data == key) printf (p->data);
}
```

Analysis of Find - 1

We would like to calculate the ***search time***, *t(n)*, it takes to find a given key in a BST with *n* nodes (or to find it is not in the BST!). First let us answer the following question:

Question: What is the *unit operation* performed to accomplish the search?

Answer: The *comparison* of the key searched with the key stored in the tree node (i.e., if (key < p->data) or if (key > p->data) or if (key == p->data)).

Hence, what we need to do to find *t(n)* is to *count, as a function of n, how many times the comparison is executed*.

Analysis of Find - 2

Now, we have converted the problem into one as in the following:

what is the average number of comparisons performed to find a given key in a BST?

or

how many nodes, on average, are visited to find a given key or to find it is not in the BST?

or, in a higher level of detail,

what is the ***average depth*** of the node that stores the key we are looking for or, in case the key is not in the BST, what is the average depth of the leaf at the end of the path our search follows through?

Analysis of Find - 3

Now, we turn back to formulating $t(n)$:

We will do the formulation for three cases:

1. Average case
2. Worst case
3. Best case

Analysis of Find: Average Case - 1

Average case:

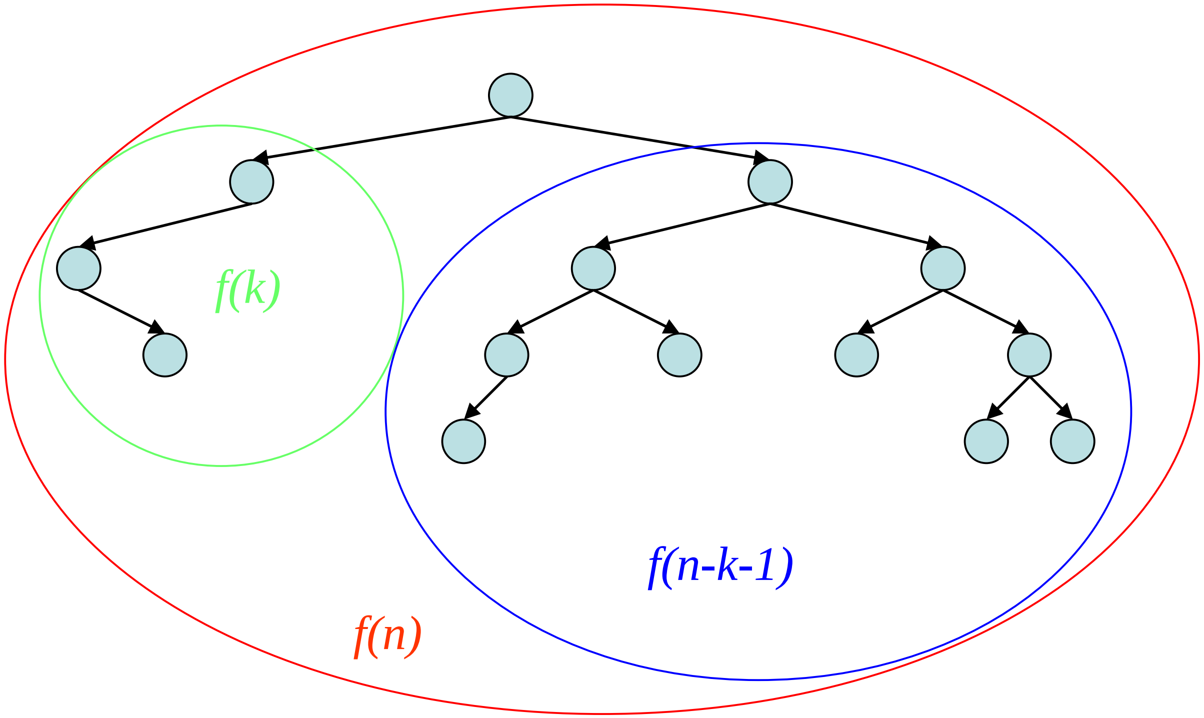
Each node has a *left and right subtree*. The search time will consist of the search time of the left subtree, right subtree and the time spent for the root in the main BST. That is, the sum of all depths of nodes in a BST (also known as the *internal path length*) will be the sum of the depths of those nodes in the left subtree, those in the right subtree and the additional contribution of the root in the main tree. We will formulate and find the *internal path length f(n)* of a BST with *n* nodes.

Assume the left and right subtrees of the root of a BST with *n* nodes have *k* and *n-k-1* nodes, respectively. Then the *depth* may be expressed as follows:

$$f(n)=f(k)+f(n-k-1)+n-1;$$

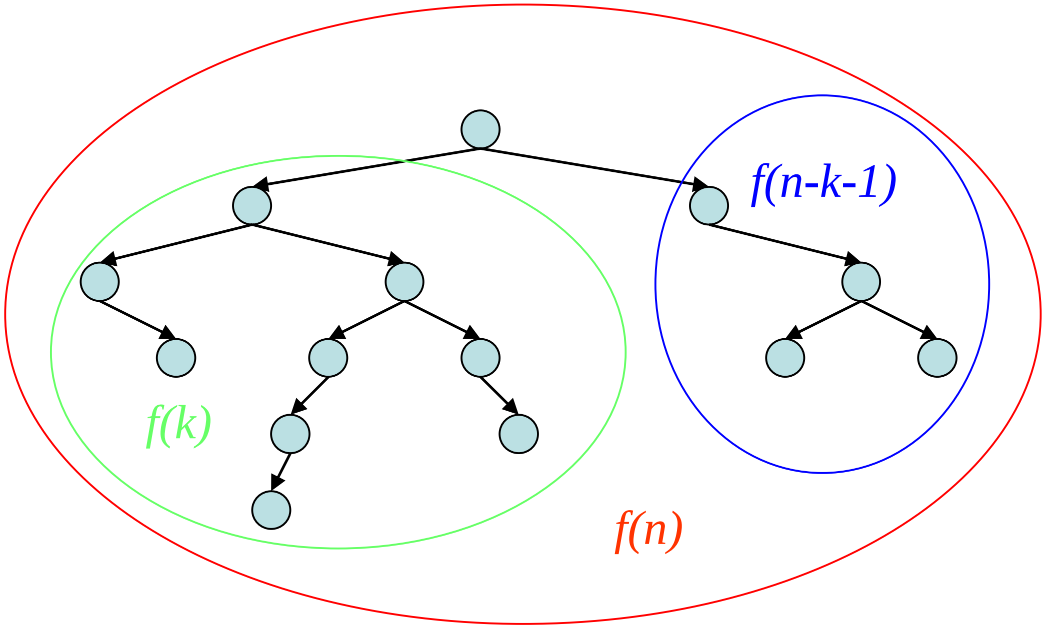
The term "*n-1*" is added to the sum of depths by the fact that all nodes in the BST are one level deeper in the main BST (because of the root of the main BST).

An example tree with *f(14)*



$$f(14) = f(3) + f(10) + 14 - 1$$
$$f(14) = 3 + 19 + 14 - 1$$
$$f(14) = 35$$
$$t(n) = f(n) + n \Rightarrow t(n) = 35 + 14 = 49$$

Another example tree with *f(14)*



$$f(14) = f(9) + f(4) + 14 - 1$$

$$f(14) = 18 + 5 + 14 - 1$$

$$f(14) = 36$$

$$t(n) = f(n) + n \quad \Rightarrow \quad t(n) = 36 + 14 = 50$$

Average Case - 2

$f(n)=f(k)+f(n\text{-}k\text{-}1)+n\text{-}1$;

k and $n\text{-}k\text{-}1$ can be any number between between 0 and $n\text{-}1$. To come up with a general solution, we can average $f(k)$ and $f(n\text{-}k\text{-}1)$:

$$f_{ave}(n) = \frac{2}{n}\sum_{k=0}^{n-1} f(k) + n - 1$$

From this point on we will drop the subscript off and use $f(n)$ to denote $f_{ave}(n)$

Average Case - 3

$$\left. f(n)=\frac{2}{n}\sum_{k=0}^{n-1} f(k)+n-1 \right| * n$$

$$nf(n)=2\sum_{k=0}^{n-1} f(k)+n(n-1); \qquad (I)$$

$$(n-1)f(n-1)=2\sum_{k=0}^{n-2} f(k)+(n-1)(n-2) \qquad (II)$$

$$nf(n)-(n-1)f(n-1)=2f(n-1)+2(n-1); (I)-(II)$$

$$nf(n)=(n+1)f(n-1)+2(n-1);$$

$$\frac{f(n)}{n+1}=\frac{f(n-1)}{n}+2\frac{(n-1)}{n(n+1)}$$

Average Case - 4

$$\frac{f(n)}{n+1} = \frac{f(n-1)}{n} + 2\frac{n-1}{n(n+1)}$$

$$\frac{f(n-1)}{n} = \frac{f(n-2)}{n-1} + 2\frac{n-2}{(n-1)n}$$

$$\vdots$$

$$\frac{f(2)}{3} = \frac{f(1)}{2} + \frac{1}{2*3}$$

$$\frac{f(1)}{2} = \frac{f(0)}{1} + 0$$

$$\frac{f(n)}{n+1} = f(0) + 2\sum_{i=1}^{n}\frac{i-1}{i(i+1)}$$

$$f(n) = (n+1)f(0) + 2(n+1)\sum_{i=1}^{n}\frac{i-1}{i(i+1)}$$

Average Case - 5

$$f(n)=(n+1)f(0)+2(n+1)\sum_{i=1}^{n}\frac{i-1}{i(i+1)};\quad \frac{i-1}{i(i+1)}=\frac{2}{i+1}-\frac{1}{i};\ f(0)=0$$

$$f(n)=-(n+1)+2(n+1)\left[2\sum_{i=1}^{n}\frac{1}{i+1}-\sum_{i=1}^{n}\frac{1}{i}\right]$$

$$f(n)=2(n+1)\left[\sum_{i=2}^{n}\frac{1}{i}-1+\frac{2}{n+1}\right]$$

$$f(n)=-2(n+1)+4+2(n+1)\underbrace{\sum_{i=2}^{n}\frac{1}{i}}_{\log(n)};\qquad \int\frac{dx}{x}=a\log x+c$$

$$f(n)=O(n+n\log(n))=O(n\log(n))$$

To find the average (per node) number of comparisons, N_{ave}, we divide *f(n)* by the number of nodes *n*: $N_{ave}= f(n)/n \Rightarrow O(\log n)$

Analysis of Find: Worst Case - 1

Worst Case BST:

The worst case BST is one with the deepest path. An *n*-node such BST is one with *n* depth levels.

Question: How many such *n*-node BSTs are there?

We will formulate and find the *internal path length f(n)* in a worst case *n*-node BST.

In such a BST one subtree of the root forms a linked list of *n-1* nodes whereas the other subtree has no nodes. Then the *depth* may be expressed as follows:

$$f(n)=f(n-1)+n-1.$$

Worst Case - 2

$$f(n) = f(n-1) + n - 1; \; f(1) = 0$$

$$CE: \quad (x-1)(x-1)^2 = 0$$

$$f(n) = c_1 + c_2 n + c_3 n^2$$

$$f(1) = c_1 + c_2 + c_3 = 0$$

$$f(2) = c_1 + c_2 2 + c_3 4 = 1$$

$$f(3) = c_1 + c_2 3 + c_3 9 = 3$$

$$\vdots$$

$$f(n) \in O(?)$$

Order of $f(n)$: $f(n)=O(n^2)$

Analysis of Find: Best Case - 1

Best Case BST:

The best case BST is one with the least tree depth. An *n*-node such BST is one with $\lfloor \log_2 n \rfloor + 1$ depth levels.

Question: How many such *n*-node BSTs are there?

We will formulate and find the *internal path length f(n)* in a best case *n*-node BST.

In such a BST both subtrees of the root have the same number of nodes (i.e., n/2 nodes). Then the *depth* may be expressed as follows:

$$f(n)=2f(n/2)+n-1.$$

Best Case - 2

$$f(n) = 2f(n/2) + n - 1;\ f(1) = 0; n = 2^k$$

$$f(k) = 2f(k-1) + 2^k - 1$$

$$CE:\quad (x-2)(x-2)\ (x-1) = 0$$

$$f(k) = c_1 2^k + c_2 k 2^k$$

$$f(n) = c_1 n + c_2 n \log_2(n)$$

$$f(1) = c_1 = 0$$

$$f(2) = 2c_1 + 2c_2 = 1 \Rightarrow c_2 = \tfrac{1}{2}$$

$$f(n) = \frac{1}{2} n \log_2(n)$$

$$f(n) \in O(?)$$

Order of $f(n)$: $f(n)=O(nlgn)$

Non-recursive FindMin Function in BSTs

```
BTNodeType *findMin (BTNodeType *p)
{// returns a pointer to node with the minimum key in the BST.
 // which one is the minimum key in a BST?

  if (p != NULL)
      while (p->left != NULL) p=p->left;
  return p;

}
```

Recursive FindMax Function in BSTs

```
BTNodeType *findMax (BTNodeType *p)
{// returns a pointer to node with the maximum key in the BST
 // which one is the maximum key in a BST?
   if (p == NULL) return NULL;
   if (p->right == NULL) return p;
   return findMax(p->right);
}
```

Recursive Insert Function in BSTs

```
BTNodeType *insert (infotype * key, BTNodeType *p)
{// returns a pointer to new node with key inserted in the BST.
    if (p == NULL) {                                    //empty tree
        p=malloc(sizeof(BTNodeType));
        if (p == NULL) return OutofMemoryError;     //no heap space left!!!
        p->data=key;
        p->left=p->right=NULL;
        return p;
    }
    else if (key < p->data)                         //tree exists, key < root
         p->left=insert(key,p->left);
    else if (key > p->data)                         //tree exists, key > root
         p->right=insert(key,p->right);
    return p;                                      //key found
}
```

Recursive Remove Function in BSTs

```
BTNodeType *Remove (infotype *key, BTNodeType *p)
{// returns a pointer to node replacing the node removed.
      BTNodeType *TempPtr;
      if (p == NULL)                                   //empty tree
            return errorMessage("empty tree");
      if (key < p->data)
        p->left=Remove(key,p->left);
      else if (key > p->data)
        p->right=Remove(key,p->right);
      else                                             //node with key found!!!
        if (p->right && p->left) {                     //if node has two children
          TmpPtr=findMin(p->right);                    //find smallest key of right subtree of node with key
          p->data=TmpPtr->data;                        //replace removed data by smallest key of right subtree
          p->right=Remove(p->data,p->right);
      }
      else {                                           //if node has less than two children
          TmpPtr=p;
          if (p->left == NULL) p=p->right;
          else if (p->right == NULL) p=p->left;
          free(TmpPtr);
      }
      return p;
}
```

Best-Average-Worst Case Access Times of a specific BST

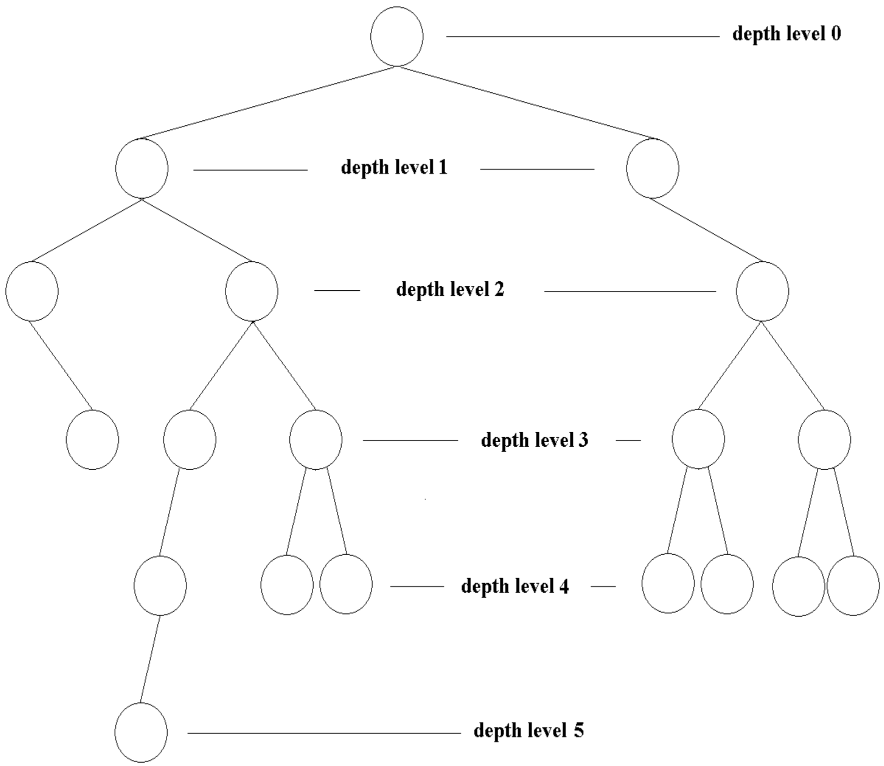
- To find a node with some specific piece of data, a search gets started from the root.
- Data at each node is compared with the key that is searched for.
- Counting up the number of comparisons for each node will render the access time.

Calculation of Access Time

- Access to some specific piece of data is achieved when the node this piece of data resides is found. To find this node, a search gets started from the root.
- Data at each node is compared with the key that is searched for.
- Counting up the number of comparisons for each node will render the access time.

Calculation of Access Time

Depth level	# of Comparisons/ node	# of Comparisons/ level
0	1	1*1=1
1	2	2*2=4
2	3	3*3=9
3	4	4*5=20
4	5	5*7=35
5	6	6*1=6
Total	Nodes: 19	75



BST Access Times	
Average	75/19=3.94
Worst	6
Best	1