Binary Search Tree (Contd.)

Binary Search Tree is a particular kind of Dynamic Set.
Data are stored inside node. Nodes are arranged as a rooted binary tree.

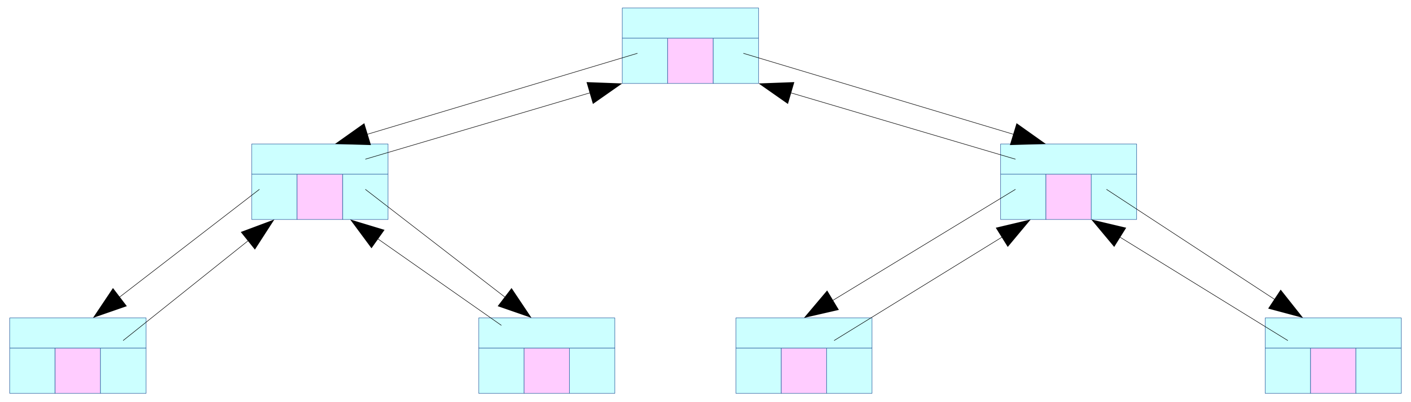
Node structure for Binary Search Tree

Pointer to parent		
Pointer to left child	Key and satellite data	Pointer to right child

node.h

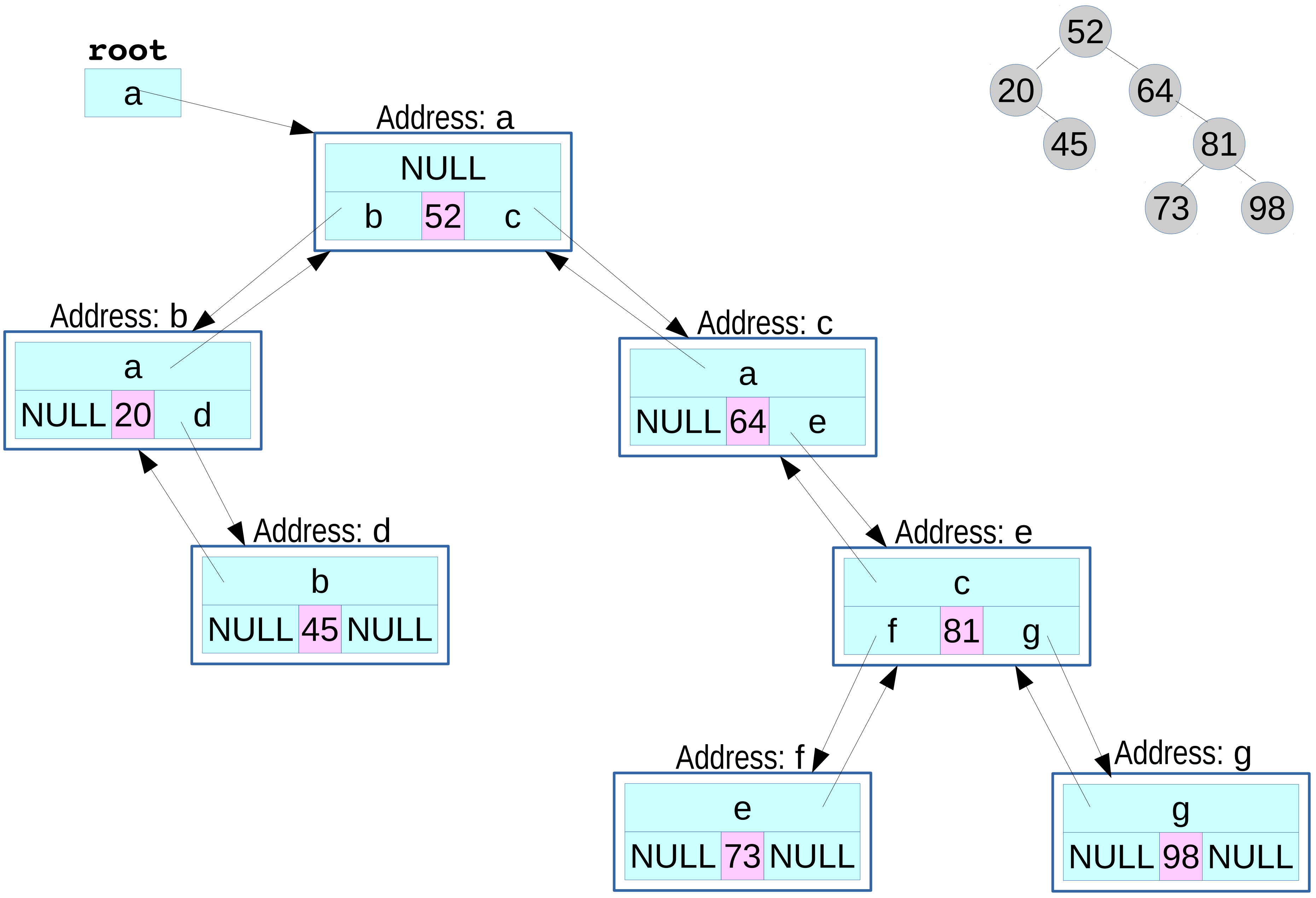
```
struct node
{
    int key;
    float sat;
    node *left, *right, *p;
};
```

- The node contains three pointers '**`left`**' , '**`right`**' and '**`p`**' which point to left child, right child and parent respectively.
  If <u>left/right child or parent does not exist</u> then the corresponding <u>pointer contains NULL</u>.
- We can design our node to contain any arbitrary collection of data.
- One of them must be the **key**. In our design we have one int type field '**`key`**' as key.
- There could be zero or more satellite data.
  In our design, we have one float type field '**`sat`**' as satellite data.



We also maintain a pointer '**`root`**' which points to the root node of the tree.
If the tree is empty then **`root`** contains NULL.

`root` is just a pointer. It is not a node itself.



Binary Search Tree Property

If node x has a left child y then, key of y < key of x.
If node x has a right child z then, key of z >= key of x.

In terms of our code the BST property looks as follows.

Let `x` be a pointer to a node.
If `x->left` is not `NULL` then `x->left->key` is less than `x->key` .
If `x->right` is not `NULL` then `x->right->key` is gtreater than or equal to `x->key` .

A class for BST

```
class bst
{
    public:
    node *root;

    bst(void);
    void Populate(int);
    void Show(void);

    void InorderTreeWalk(node *);

    node *TreeSearch(node *, int);
    node *IterativeTreeSearch(node *, int);
    node *TreeMinimum(node *);
    node *TreeMaximum(node *);
    node *TreeSuccessor(node *);
    node *TreePredecessor(node *);

    void TreeInsert(node *);
    void Transplant(node *, node *);
    void TreeDelete(node *);
};
```

To be implemented in today's lab.
(You also need **TreeMinimum** because
it is called by TreeDelete())

For Problem 3 we have added two new member functions Size() and Height()
(and removed whatever is not required)

```
class bst
{
   public:
   node *root;

   bst(void);
   InorderTreeWalk(node *);

   void TreeInsert(node *);

   int Size(node *);
   int Height(node *);
};
```

Tasks that the member functions are supposed to do

```
void TreeInsert(node *);
```

Given a pointer x to a node (which is not yet part of the tree but contains key and satellite data), `TreeInsert(x)` inserts the node into the tree.

```
void TreeDelete(node *);
```

Given a pointer x to a node in the tree, `TreeDelete(x)` deletes the node from the tree.

```
void Transplant(node *, node *);
```

Given two pointers u and v to two nodes in the tree, `Transplant(u,v)` transplants the subtree rooted at u with the subtree rooted at v. (See Cormen for a detailed understanding)

For problem 3

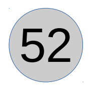
```
int Size(node *);
```

Given a pointer x to a node in the tree, `Size(x)` returns the total number of nodes nodes in the subtree rooted at x.

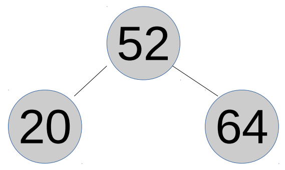
```
int Height(node *);
```

Given a pointer x to a node in the tree, `Height(x)` returns the height of the subtree rooted at x. (height of empty tree is -1, height of a tree containing only one node is 0)

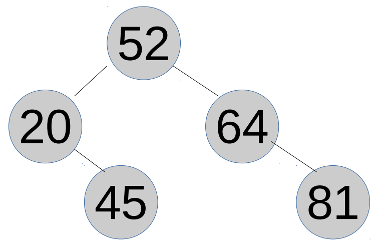
Example:



Height 0

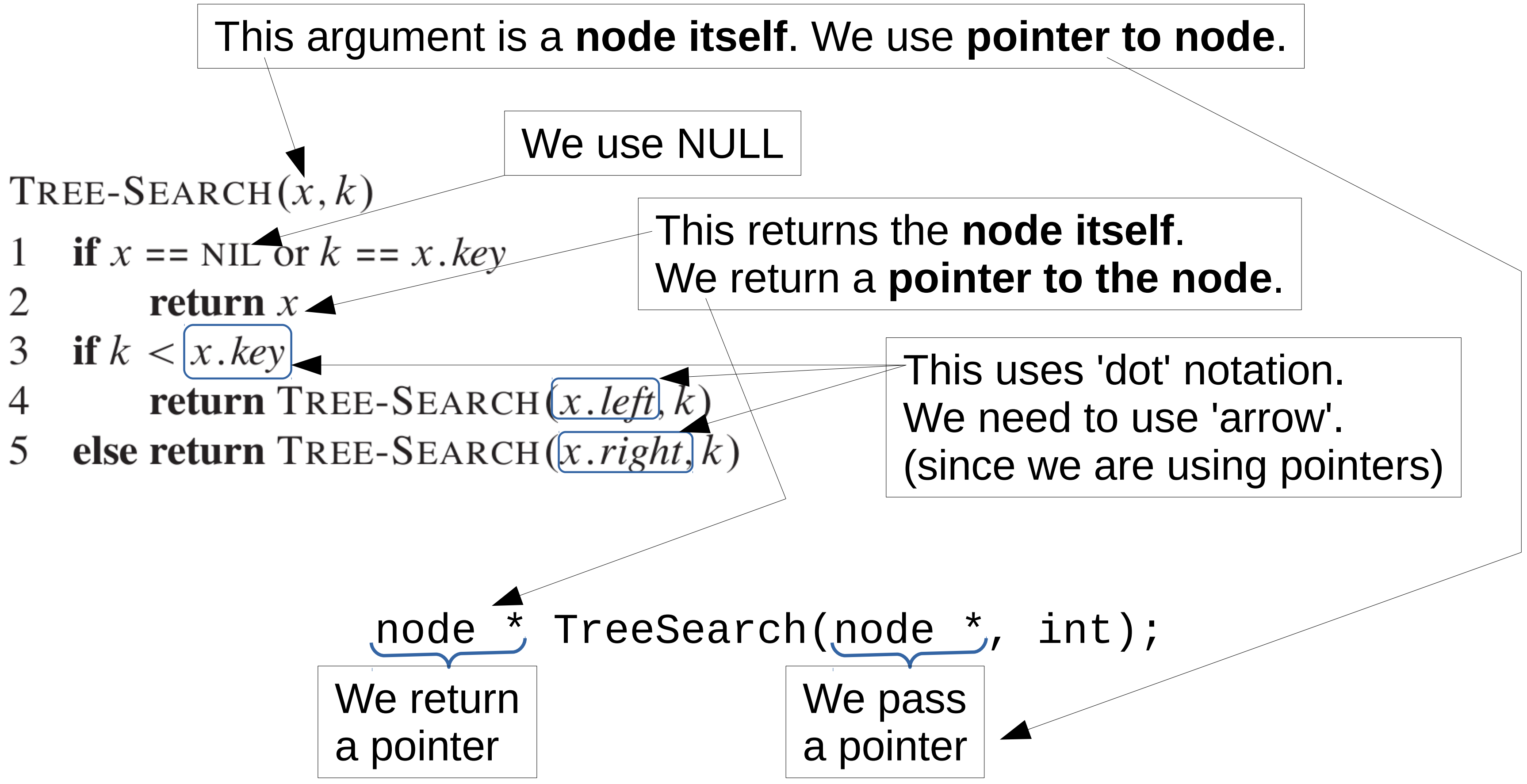


Height 1



Height 2

Difference between the pseudocode (in Cormen) and our C++ code



```
TREE-SEARCH(x, k)
1  if x == NIL or k == x.key
2      return x
3  if k < x.key
4      return TREE-SEARCH(x.left, k)
5  else return TREE-SEARCH(x.right, k)
```

```
node * TreeSearch(node *, int);
```

This argument is a **node itself**. We use **pointer to node**.
We use NULL
This returns the **node itself**. We return a **pointer to the node**.
This uses 'dot' notation. We need to use 'arrow'. (since we are using pointers)
We return a pointer
We pass a pointer

Difference between the pseudocode (in Cormen) and our C++ code (contd.)

This takes the tree as a parameter

```
TRANSPLANT($T, u, v$)
1  if $u.p == $ NIL
2      $T.root = v$
3  elseif $u == u.p.left$
4      $u.p.left = v$
5  else $u.p.right = v$
6  if $v \neq$ NIL
7      $v.p = u.p$
```

For us Transplant() is a member function.
We do not neen to pass the tree as a parameter.

Hence we have the declaration.

`void Transplant(node *, node *)`

And we should use **`root`** in stead of **`T.root`**

(These apply for TreeInsert() and TreeDelete() also)

One special note about TreeDelete()
The pseudocode does not explicitly mention about deallocation of space.
Since we are using dynamic allocation at the time of creating a node,
we need to deallocate the space explicitly during TreeDelete().

Helper functions (insert() and remove())

Notice that `TreeInsert(x)` takes a pointer to a node as an argument. It assumes that the node is already constructed and the values for key and satellite data are set, but the node has not yet been inserted into the tree.

The Job of `TreeInsert(x)` is to just insert the node.

So we consider the function **`void insert(bst&, int, float);`**
It is NOT a member function
When we call **`insert(t,k,s)`**
`t` is supposed to be an object of class bst. And it is passed by reference (notice the &),
`k` is supposed to be a key (hence an integer), and
`s` is supposed to be the corresponding satellite data (hence a float).

`insert(t,k,s)` is supposed to create a node (using dynamic allocation) with key k and satellite data s. And then insert the node (using TreeInsert()) into the tree t.

Similarly,
`TreeDelete(x)` takes a pointer to a node as an argument and deletes that node from the tree.

So we consider the function **`void remove(t, int);`** which is **NOT a member function**.

When we call **`remove(t,k)`** , t is supposed to be an object of class bst (passe by reference) and k is supposed to be a key. And **`remove(t,k)`** is supposed to delete the node with key k (if there is any such node) from tree t. (you may assume that the keys are unique)

You also need to implement the function **`void removelt(bst&,float);`**

It is **not a member function**.

The first argument is an object of class bst and it is passed by reference.
The second argument is a float.

`removelt(t,x)` should delete, from tree t, all the nodes whose satellite data is less than x.

Algorithms

```
TREE-INSERT($T, z$)
$y = $ NIL
$x = T.root$
while $x \neq$ NIL
    $y = x$
    if $z.key < x.key$
        $x = x.left$
    else $x = x.right$
$z.p = y$
if $y ==$ NIL
    $T.root = z$
elseif $z.key < y.key$
    $y.left = z$
else $y.right = z$
```

```
TREE-MINIMUM($x$)
while $x.left \neq$ NIL
    $x = x.left$
return $x$
```

```
TRANSPLANT($T, u, v$)
if $u.p ==$ NIL
    $T.root = v$
elseif $u == u.p.left$
    $u.p.left = v$
else $u.p.right = v$
if $v \neq$ NIL
    $v.p = u.p$
```

```
TREE-DELETE($T, z$)
if $z.left ==$ NIL
    TRANSPLANT($T, z, z.right$)
elseif $z.right ==$ NIL
    TRANSPLANT($T, z, z.left$)
else $y = $ TREE-MINIMUM($z.right$)
    if $y.p \neq z$
        TRANSPLANT($T, y, y.right$)
        $y.right = z.right$
        $y.right.p = y$
    TRANSPLANT($T, z, y$)
    $y.left = z.left$
    $y.left.p = y$
```

Figure out algo/code for the following

Size()
Height()
insert()
remove()
removeIt()

Figure out algorithm / code for: Size(), Height(), insert(), remove() and removeIt()

Hint

- Study the algorithm/code for inorder tree walk. Understand how recursion is used.
- If required define other functions.
- If required use pointer to pointer or reference to pointer.
- Brush up your knowledge about 'reference variable' and how there are used as arguments of functions.