Classes can have public, private or protected members.

Member function of a class can access all members of that class.

User of the class can access only public members of the class.

```
class B
{
    public:
    void f(void);
    int x1;

    protected:
    int x2;

    private:
    int x3;
};
```

```
void B::f(void)
{
  x1=0;    OK
  x2=0;    OK
  x3=0;    OK
}
```

```
main()
{
    B b;
    b.f();       OK

    b.x1 = 0;    OK
    b.x2 = 0;    ERROR
    b.x3 = 0;    ERROR
}
```

In C++ there are 3 types of inheritance - public, protected and private.

For all of them, **derived class never inherits private member of base class**.

Derived class inherits public and protected members of base class.

Base class

```
class B
{
 public:
 int x1;
 protected:
 int x2;
 private:
 int x3;
};
```

public or **private** or **protected**

Derived class

```
class D:...... B
{
 public:
 void g(void);
};
```

```
main()
{
 B b;
 b.x1=0; OK
 b.x2=0; ERROR
 b.x3=0; ERROR
}
```

```
void D::g(void)
{
 x1=0; OK
 x2=0; OK
 x3=0; ERROR
}
```

Public members: User **can** access. **Can** be inherited.

Protected members: Only member functions can access. User **cannot** access. **Can** be inherited.

Private member: Only member functions can access. User **cannot** access. **Cannot** be inherited.

Class B has the members: x1 (public) , x2 (protected) , x3 (private)

Class D is derived from B (there are 3 types of inheritance)

Class D has the members: x1 , x2   ,~~x3~~

Is it public ?
Or private ?
Or protected ?

Is it public ?
Or private ?
Or protected ?

Difference between public, protected and private inheritance

In C++ there are 3 types of inheritance - public, protected and private.

For all of them, derived class never inherits private member of base class.

Derived class inherits public and protected members of base class.

Base class	Public Inheritance	Protected Inheritance	Private Inheritance

`class B`
`{`
` public:`
` void f(void);`
` int x1;`
` protected:`
` int x2;`
` private:`
` int x3;`
`};`
	`class D:public B`
	`{`
	` .......`
	`};`
	
	x1 becomes a public member of D
	
	x2 becomes a protected member of D
	
	x3 is not inherited
		`class D:protected B`
		`{`
		` ........`
		`};`
		
		x1 and x2 become protected members of D
		
		x3 is not inherited
			`class D:private B`
			`{`
			` .........`
			`};`
			
			x1 and x2 become private members of D
			
			x3 is not inherited

Public Inheritance: Public members of base class become Public members of derived class and Protected members of base class become Protected members of derived class

Protected Inheritance: Public and Protected members of base class become Protected members of derived class

Private Inheritance: Public and Protected members of base class become Private members of derived class

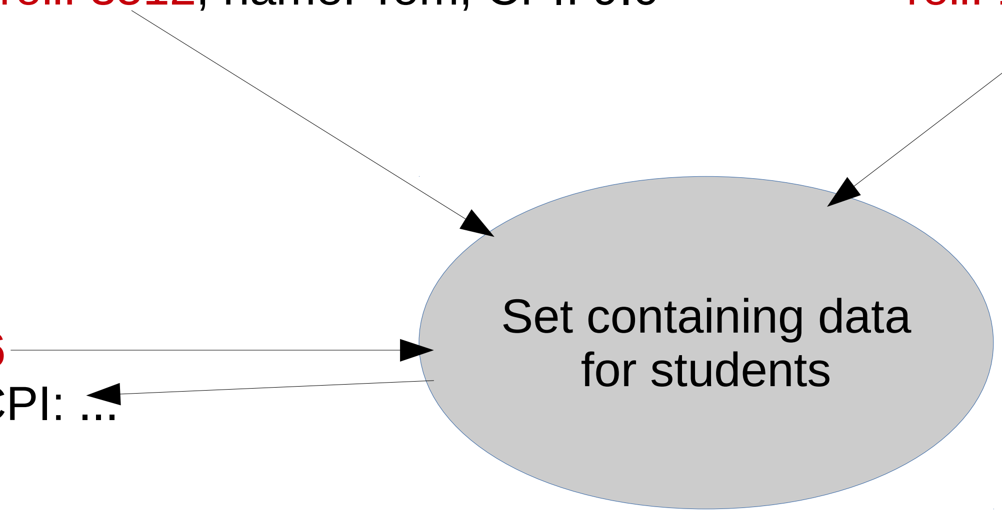
Binary Search Tree

Dynamic Set

Insert
Insert following data
roll: 3512, name: Tom, CPI: 9.0

Delete
Delete all data for
roll: 1128

Search
Get all data about roll: 5126
Search returns, name: ..., CPI: ...



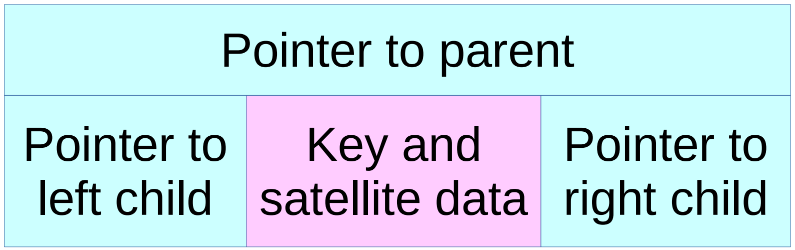
Dynamic Set is a data structure which can store a set a data and allows us to **Insert**, **Delete** and **Search** for data.

Data contains a key. During Insert, Delete and Search we must always specify the **key**.
In our example, the key is roll.

Data other than the key is called satellite data.
In our example, name and CPI are satellite data.

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

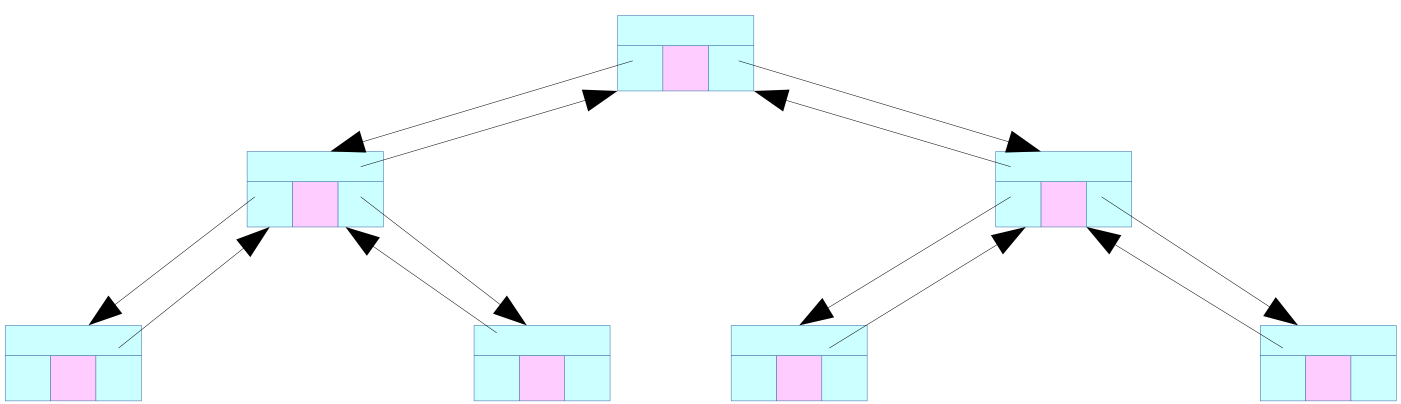
Node structure for Binary Search Tree



node.h

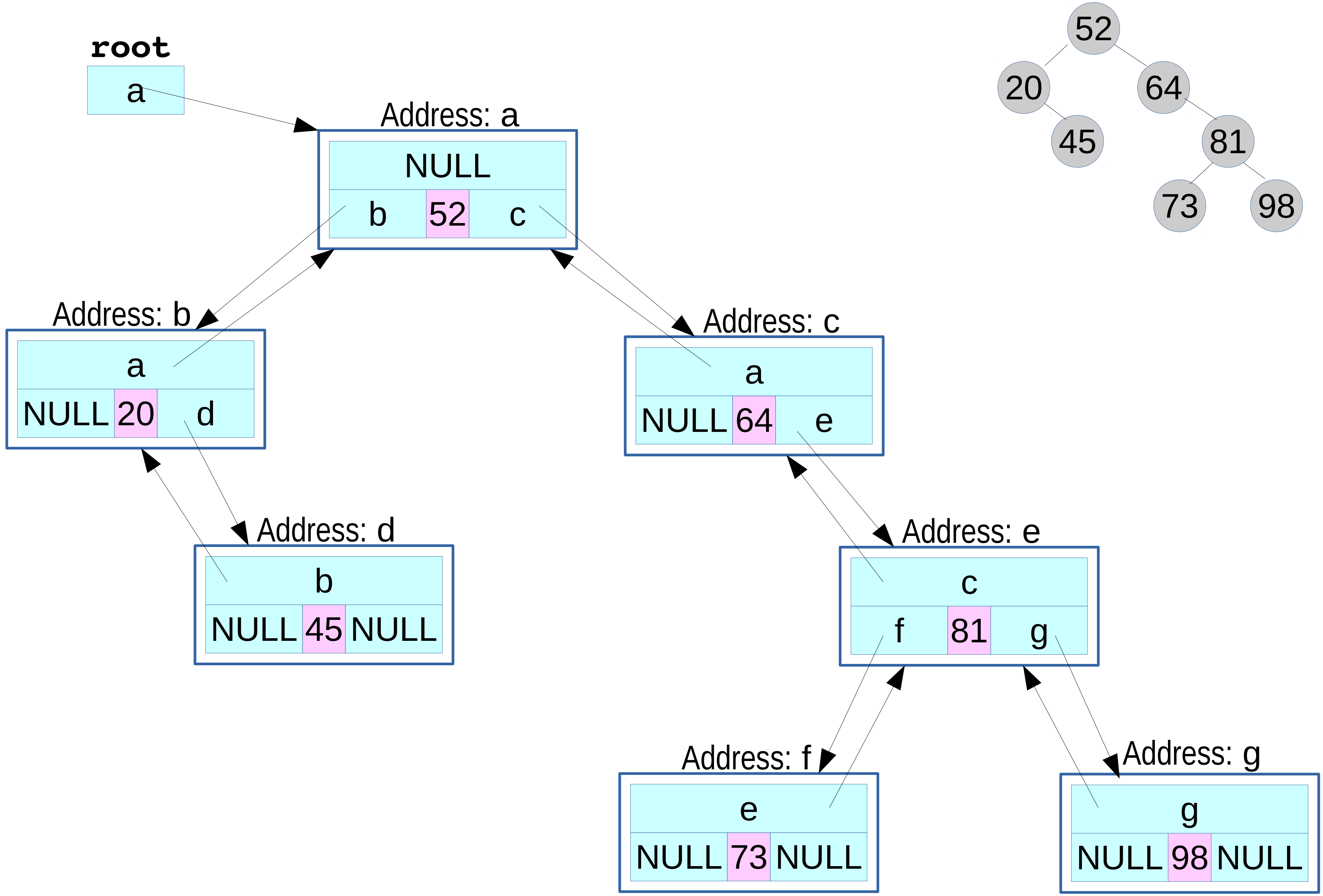
```
struct node
{
    int key;
    float sat;
    node *left, *right, *p;
};
```

- 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 <u>just a pointer</u>. It is <u>not a node itself</u>.



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 *);
};
```

In today's lab you will implement only these

Tasks that the member functions are supposed to do

```
void InorderTreeWalk(node *);
```

Given a pointer x to a node, performs an inorder walk on the subtree rooted at that node.

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

Given a pointer x to a node and a key k, searches for the key k inside the subtree rooted at x.
If found, returns a pointer to the node containing the key. Otherwise returns NULL.

```
node *TreeMinimum(node *);
node *TreeMaximum(node *);
```

Given a pointer x to a node, searches for the max/minimum key inside the subtree rooted at x.
If found, returns a pointer to the node containing the max/minimum key. Otherwise returns NULL.

```
node *TreeSuccessor(node *);
node *TreePredecessor(node *);
```

Given a pointer x to a node, searches for the successor/predecessor of that node.
If found, returns a pointer to the successor/predecessor node. Otherwise returns NULL.

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

This argument is a **node itself**. We use **pointer to node**.

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

This returns the **node itself**.
We return a **pointer to the node**.

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

We return a pointer

We pass a pointer

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

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)

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

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

We return
a pointer

We pass
a pointer

Algorithms

ITERATIVE-TREE-SEARCH(x, k)

```
while x ≠ NIL and k ≠ x.key
    if k < x.key
        x = x.left
    else x = x.right
return x
```

TREE-MAXIMUM(x)

```
while x.right ≠ NIL
    x = x.right
return x
```

TREE-MINIMUM(x)

```
while x.left ≠ NIL
    x = x.left
return x
```

TREE-SUCCESSOR(x)

```
if x.right ≠ NIL
    return TREE-MINIMUM(x.right)
y = x.p
while y ≠ NIL and x == y.right
    x = y
    y = y.p
return y
```

TREE-PREDECESSOR(x)

Figure it out yourself !