7.5 Binary search tree

7.5.1 Definition

7.5.2 Searching a binary search tree

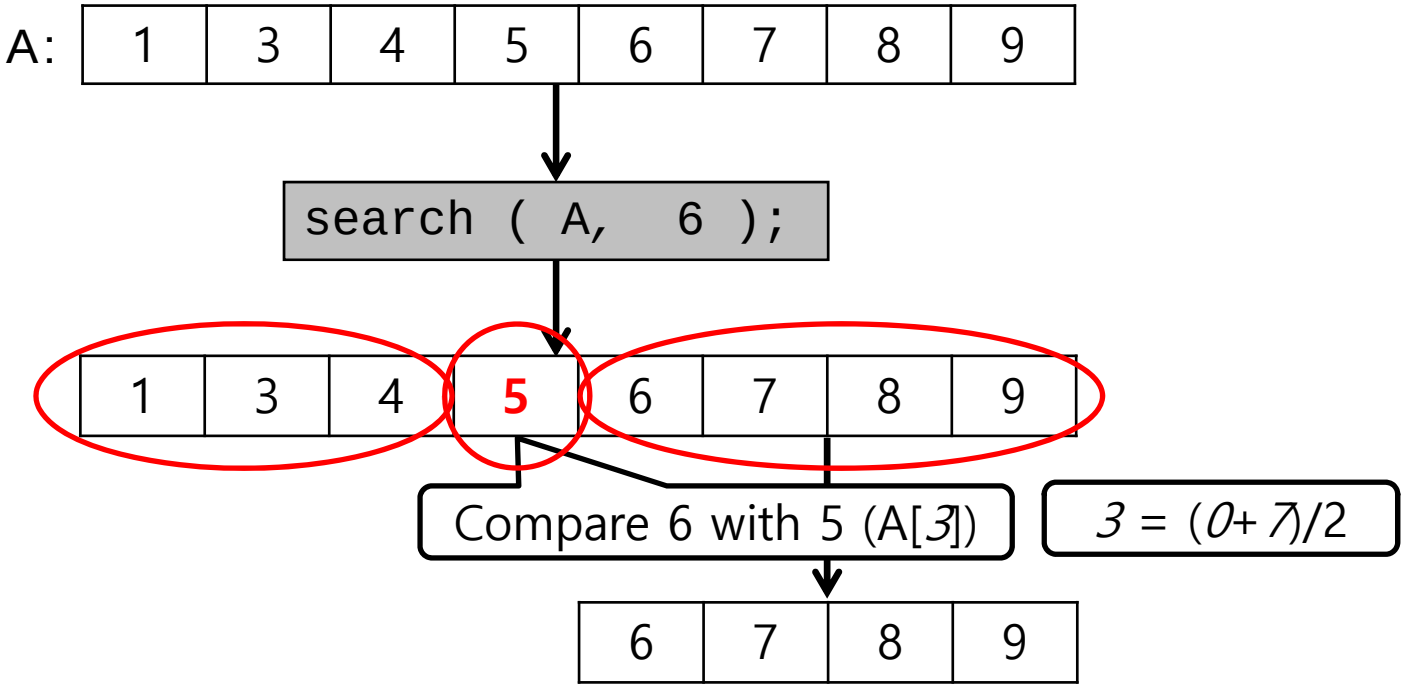
7.5.3 Inserting into a binary search tree

7.5.4 Deletion from a binary search tree

7.5.5 Time complexity on a binary search tree

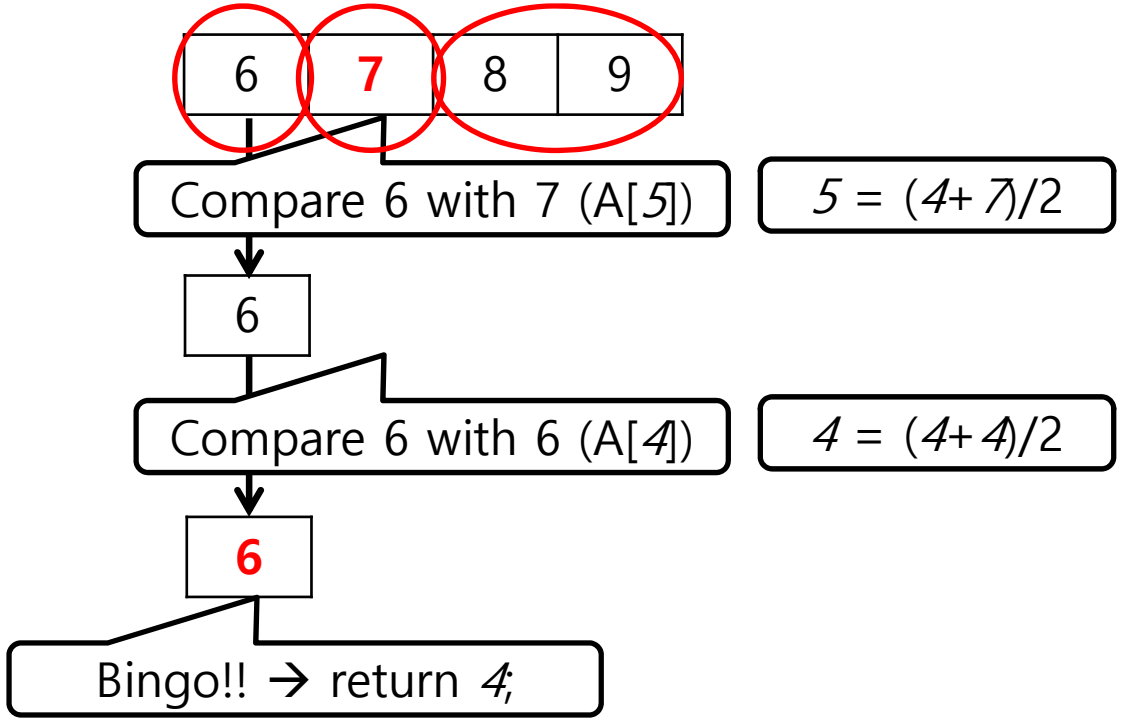
7.5.1 Definition

- Recall “binary search”
  - select the **middle** of the array and divide the array by half (**left** & **right**)



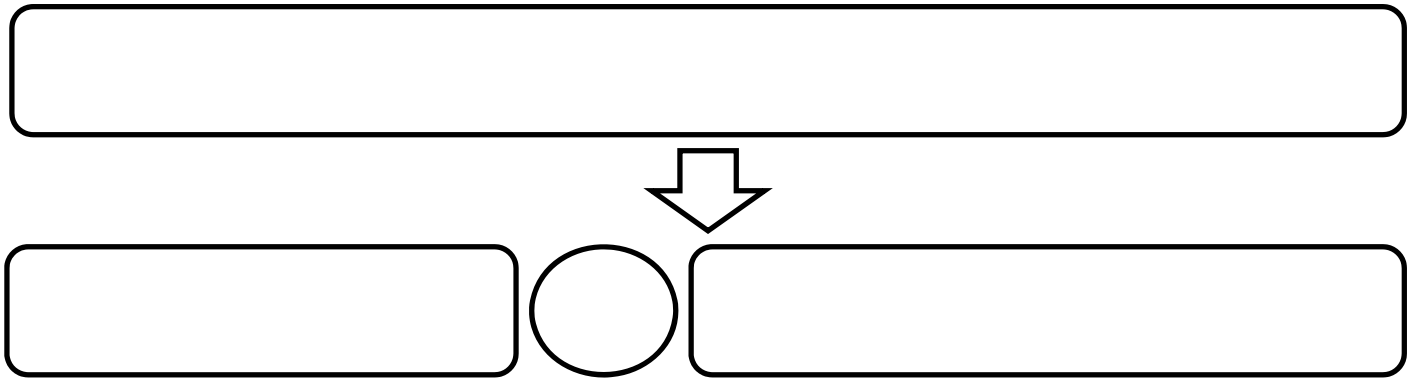
7.5.1 Definition

- Recall “binary search”
  - select the **middle** of the array and divide the array by half (**left** & **right**)



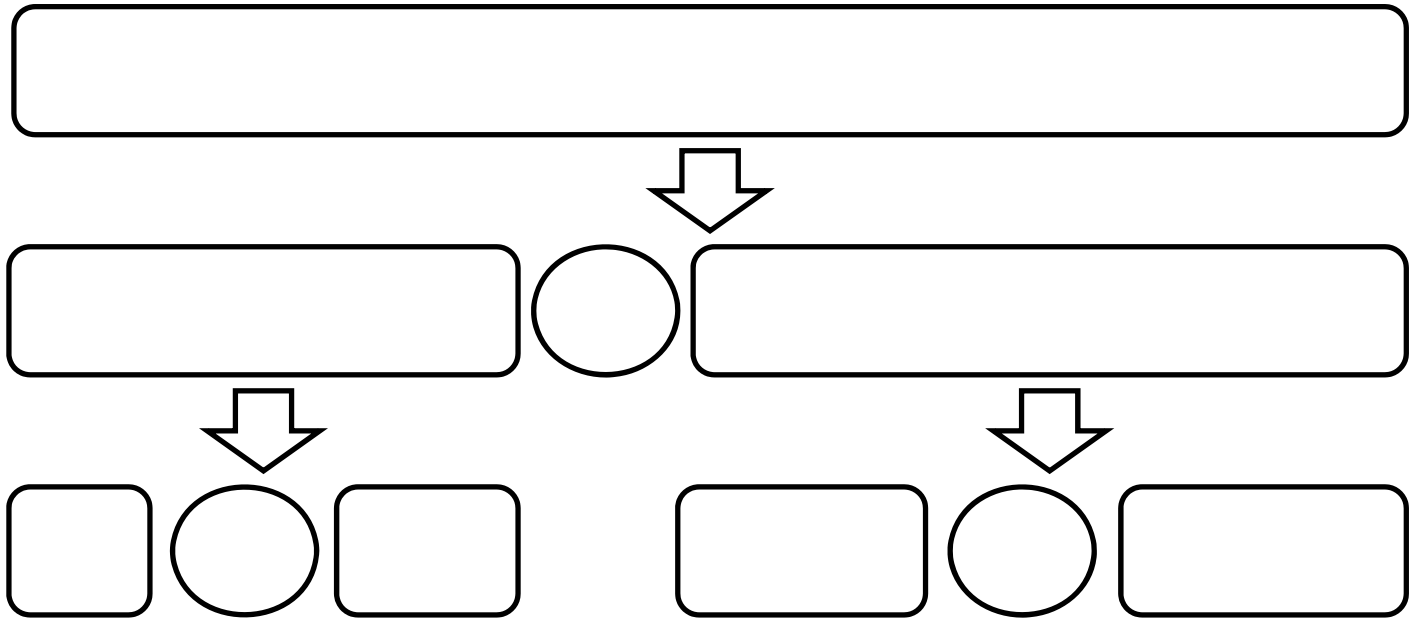
7.5.1 Definition

- Recall “binary search”
  - select the **middle** of the array and divide the array by half (**left** & **right**)



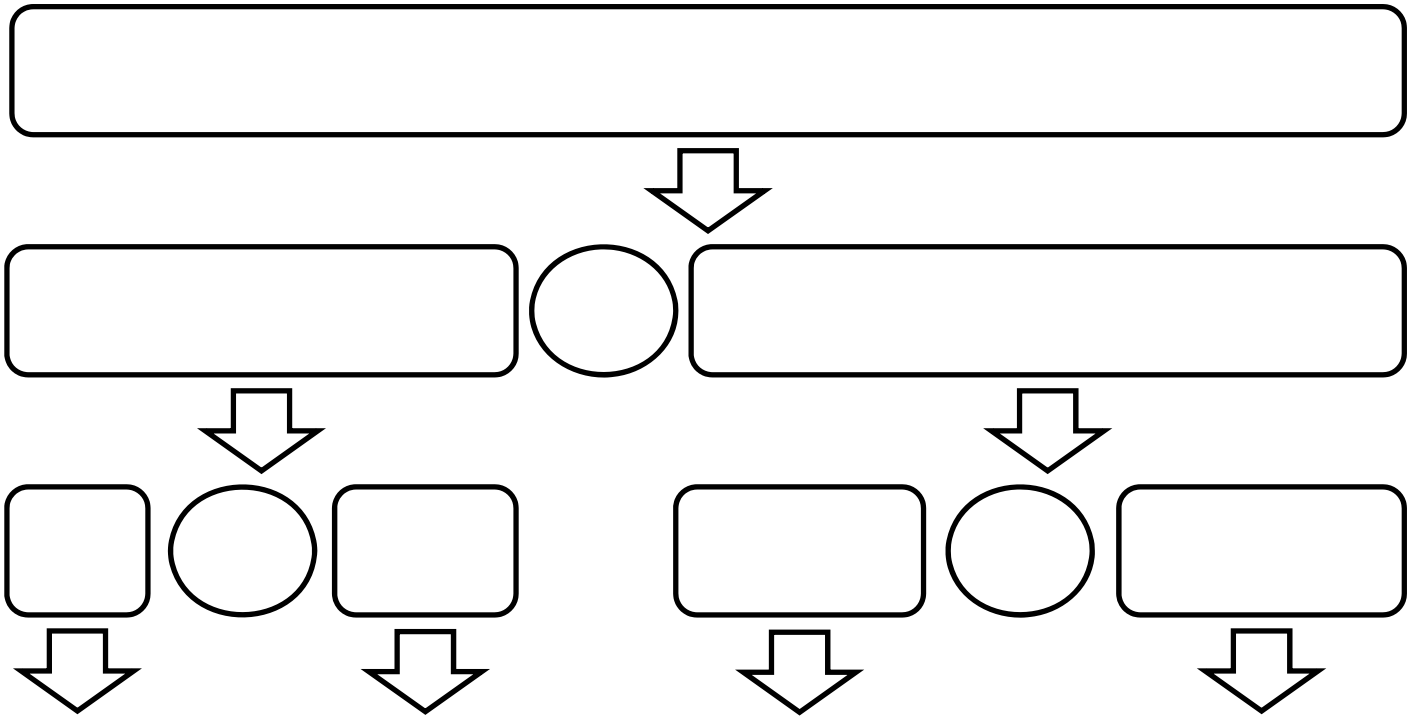
7.5.1 Definition

- Recall “binary search”
  - select the **middle** of the array and divide the array by half (**left** & **right**)



7.5.1 Definition

- Recall “binary search”
  - select the **middle** of the array and divide the array by half (**left** & **right**)



7.5.1 Definition

- A structure that supports binary search
  - Recursive structure
    - structure →
      (left structure) + middle + (right structure)
    - tree →
      (left subtree) + root node + (right subtree)
  - Comparison
    - all values in the left structure < middle
    - all values in the right structure > middle

7.5.1 Definition

- Binary search tree
  - A binary tree (may be empty)
  - Satisfies the following properties

    (1) Each node has **exactly one key** and the keys in the tree are distinct

    (2) The keys in the **left** subtree are **smaller** than the key in the root

    (3) The keys in the **right** subtree are **larger** than the key in the root

    (4) The left and right subtrees are also **binary search tree**

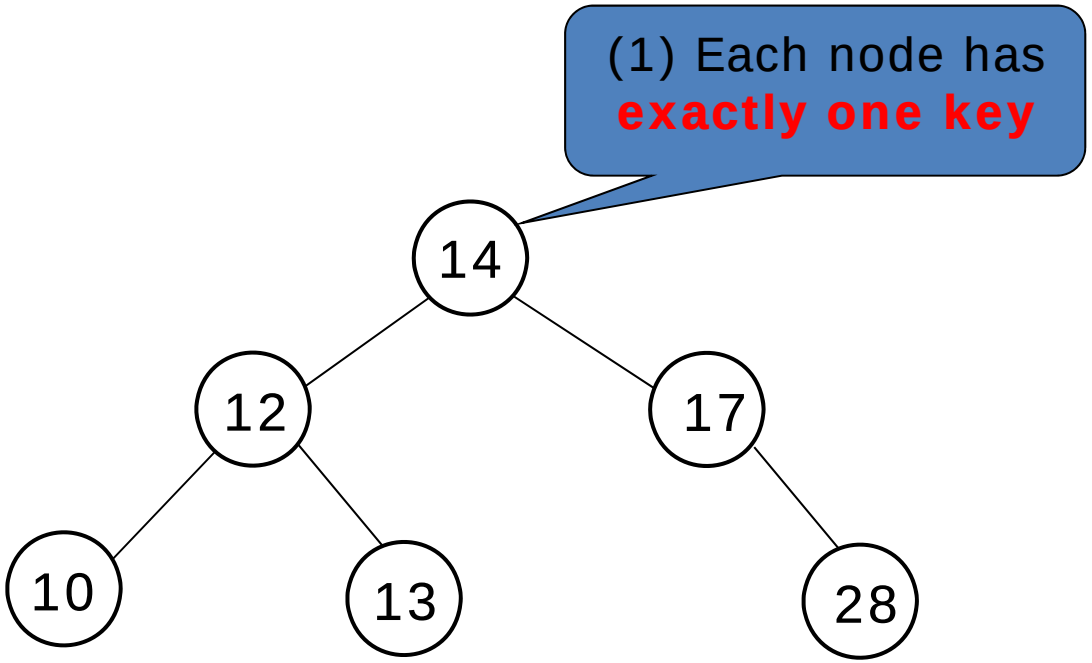
7.5.1 Definition

- Data structures for efficient search

Data structure		**Insert**	**Delete**	**Search**	**Get max (Pop)**	**Remove max (Top)**
Array	Unsorted	O(1)	O(n)	O(n)	O(n)	O(n)
	Sorted	O(n)	O(n)	**O(log n)**	**O(1)**	O(n)
Linked list	Unsorted	O(n)	O(n)	O(n)	O(n)	O(n)
	Sorted	O(n)	O(n)	O(n)	O(1)/O(n)	O(1)/O(n)
Binary search tree	*BC*					
	WC					
Heap						
Hash table						

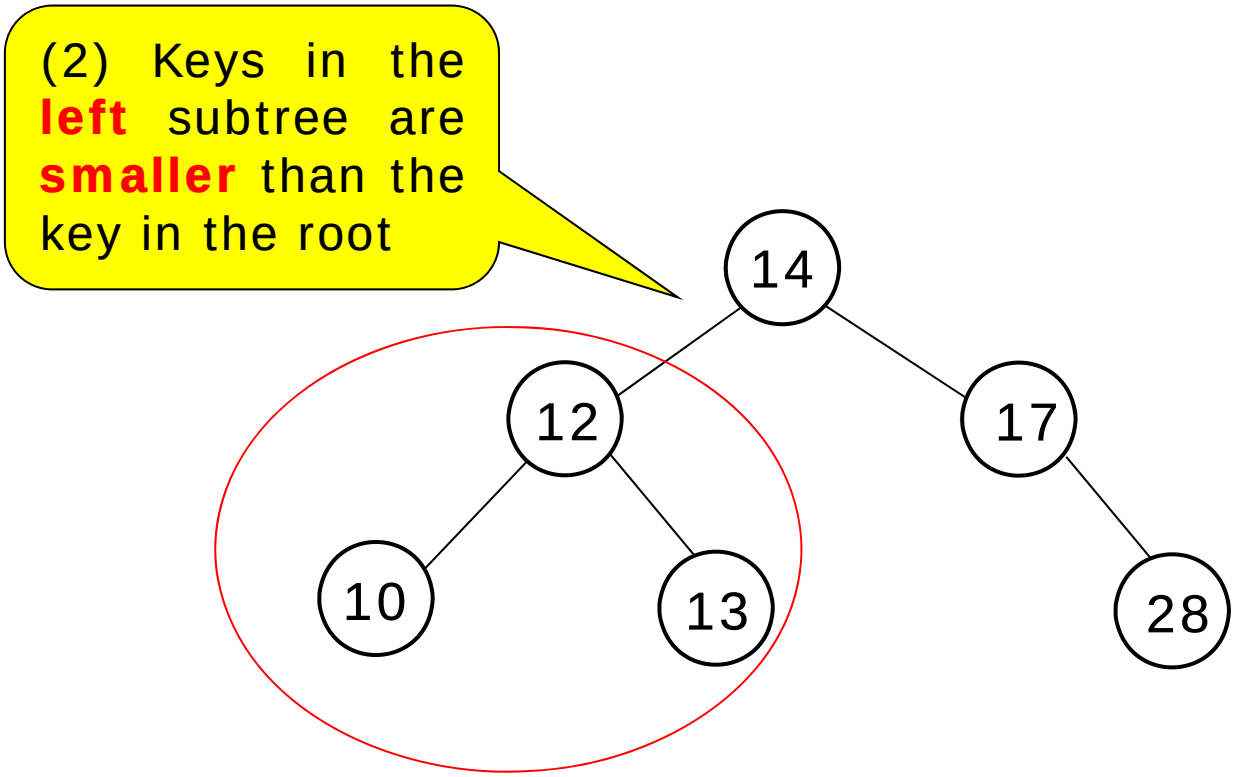
7.5.1 Definition

- Binary search tree



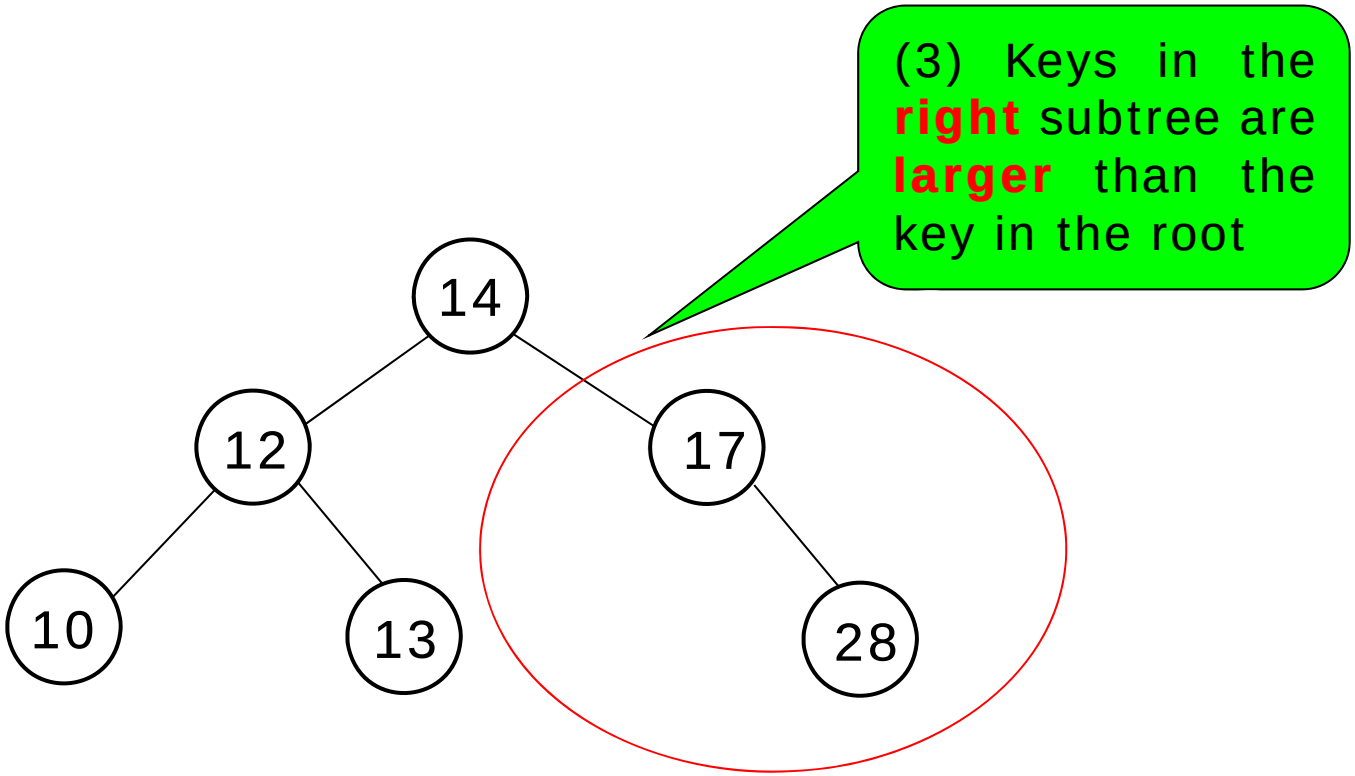
7.5.1 Definition

- Binary search tree



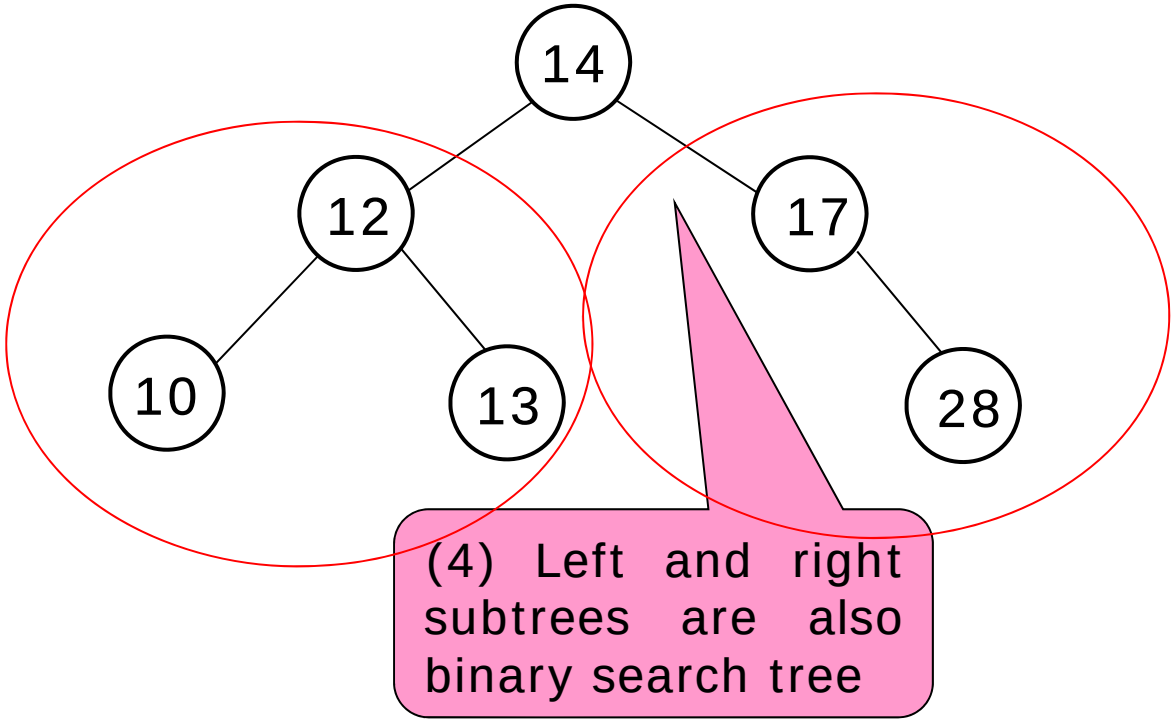
7.5.1 Definition

- Binary search tree



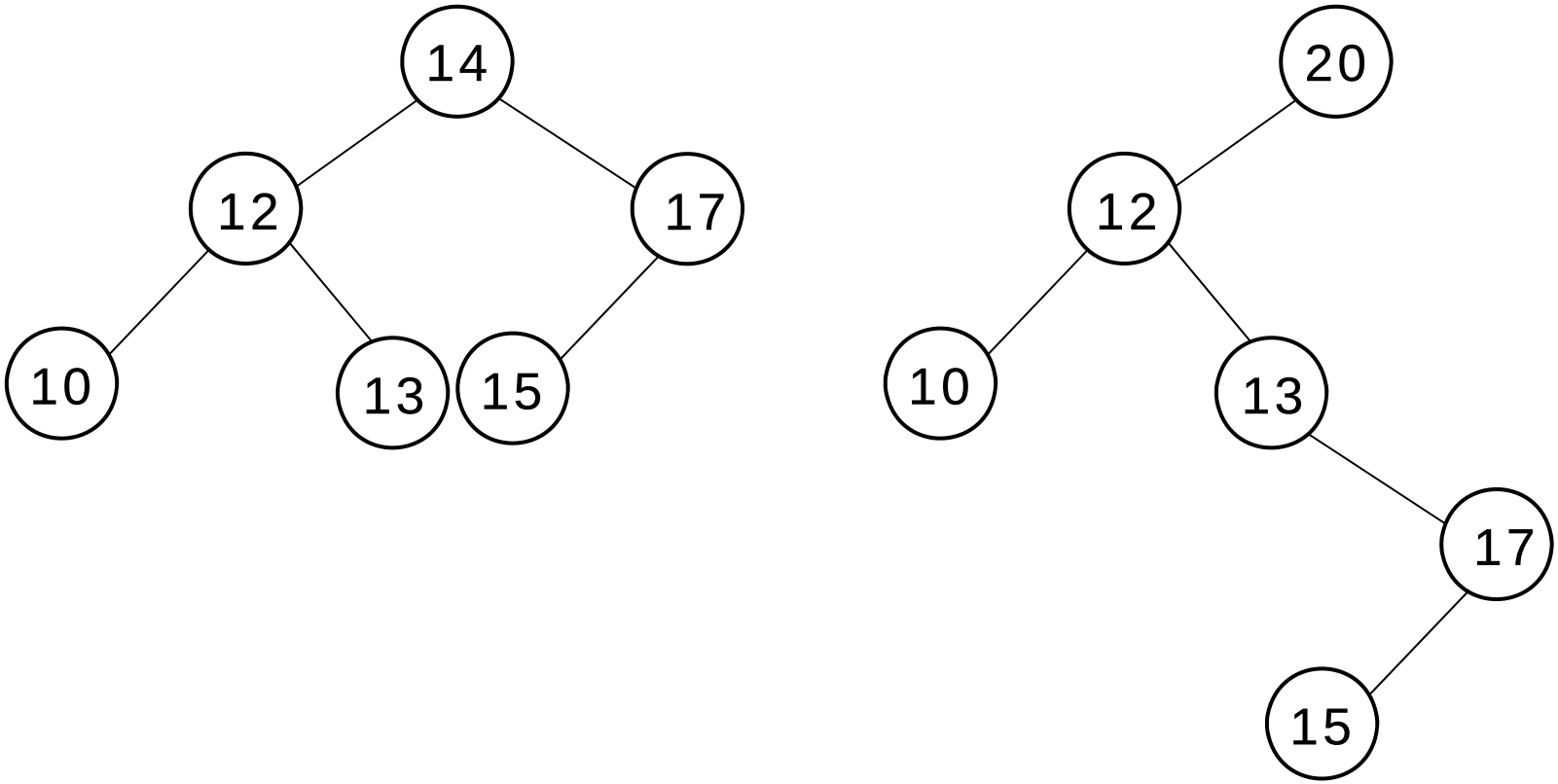
7.5.1 Definition

- Binary search tree



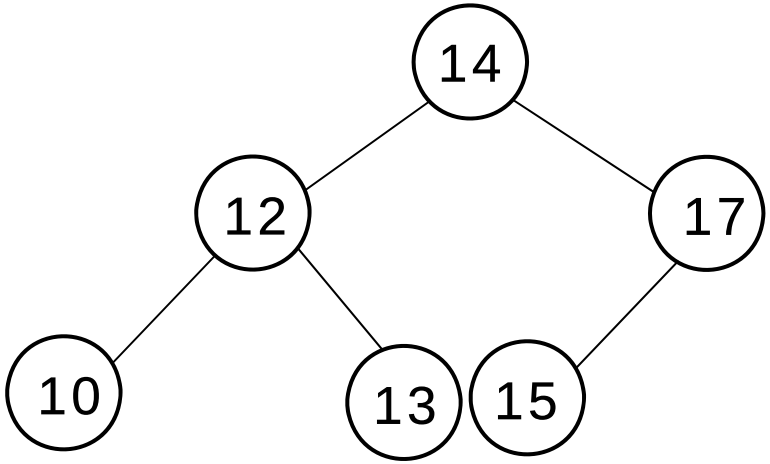
7.5.1 Definition

- Binary search trees



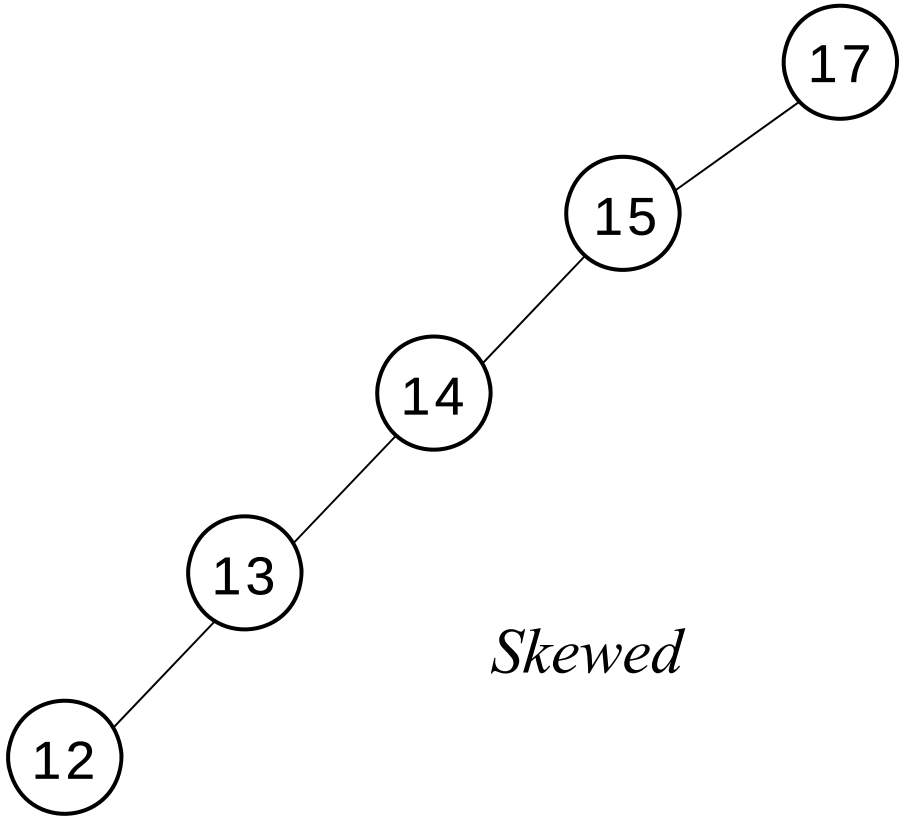
7.5.1 Definition

- Binary search trees (good and bad)



Balanced

→ $|depth(left) - depth(right)| \leq 1$



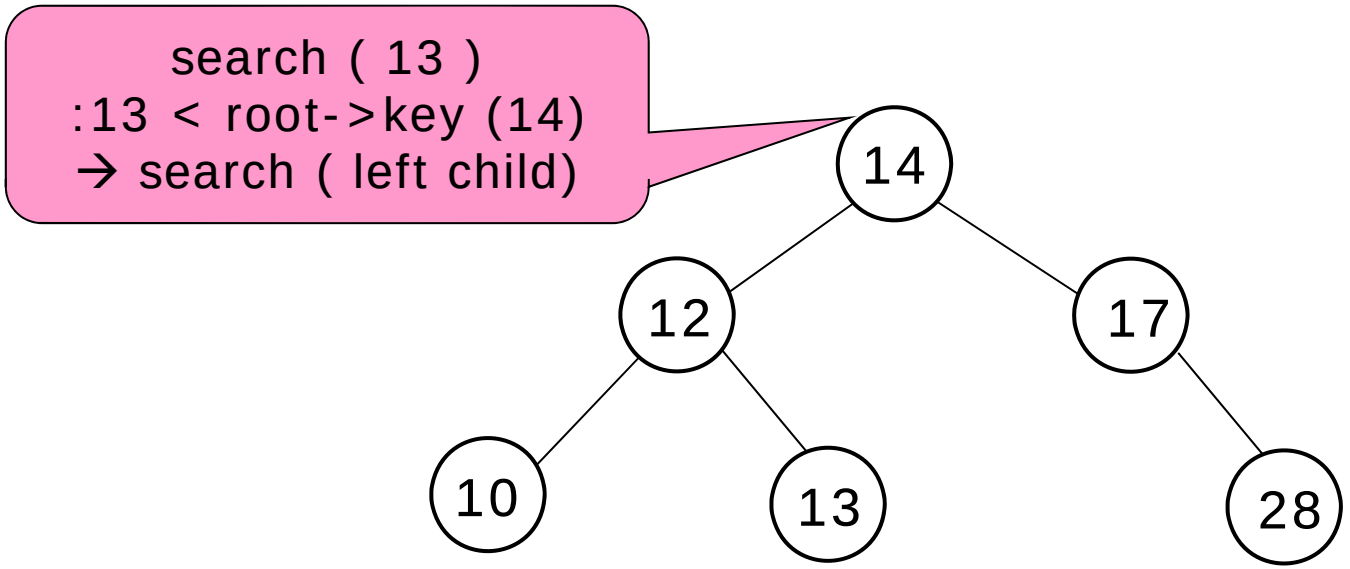
Skewed

7.5.2 Search

- Given a binary search tree, find a node whose key is k

```
element node::search (KEY key )
```

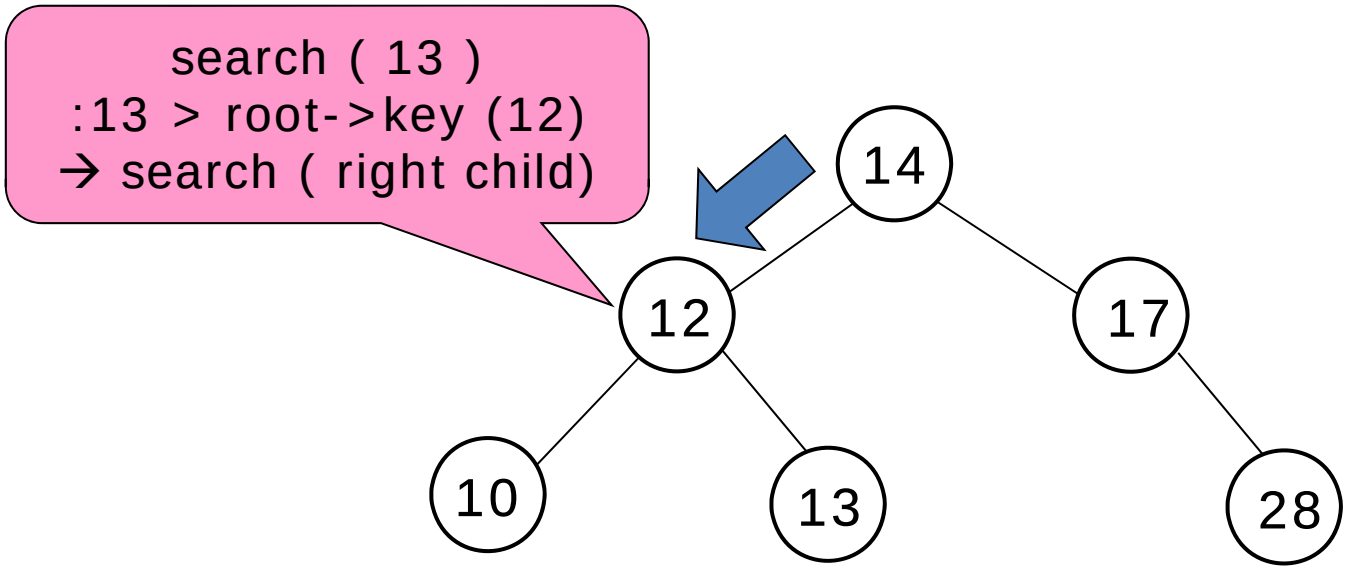
```
root->search ( 15 );
```



7.5.2 Search

- Given a binary search tree, find a node whose key is k

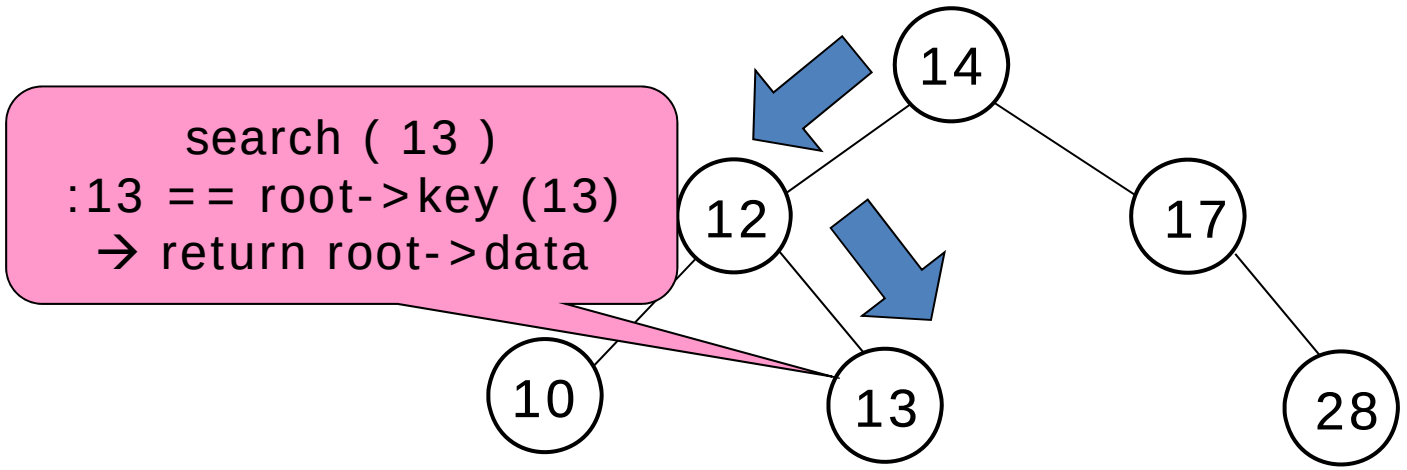
```
element node::search (KEY key )
```



7.5.2 Search

- Given a binary search tree, find a node whose key is k

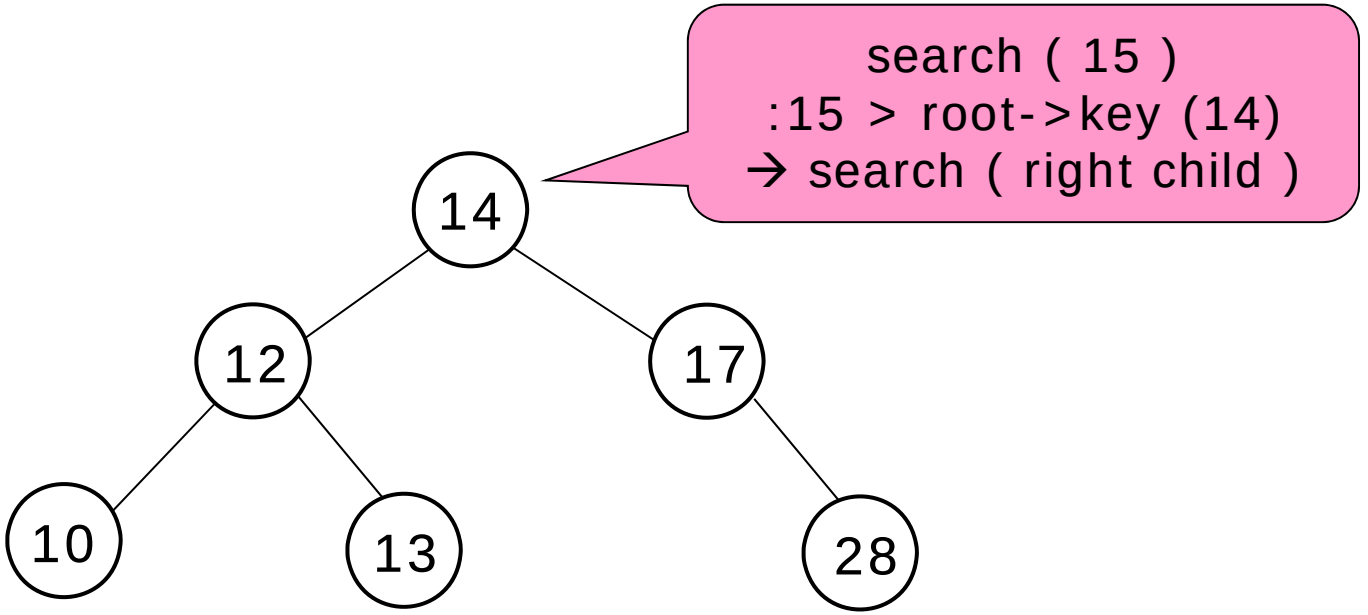
```
element node::search (KEY key )
```



7.5.2 Search

- Given a binary search tree, find a node whose key is k

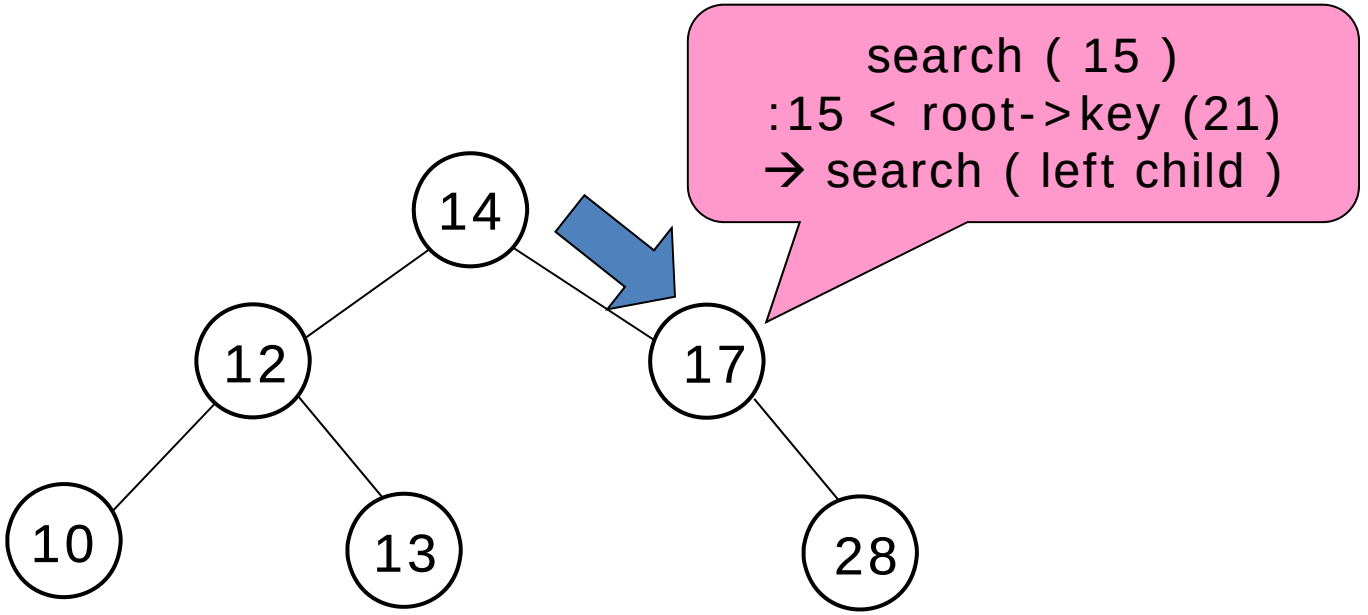
```
element node::search (KEY key )
```



7.5.2 Search

- Given a binary search tree, find a node whose key is k

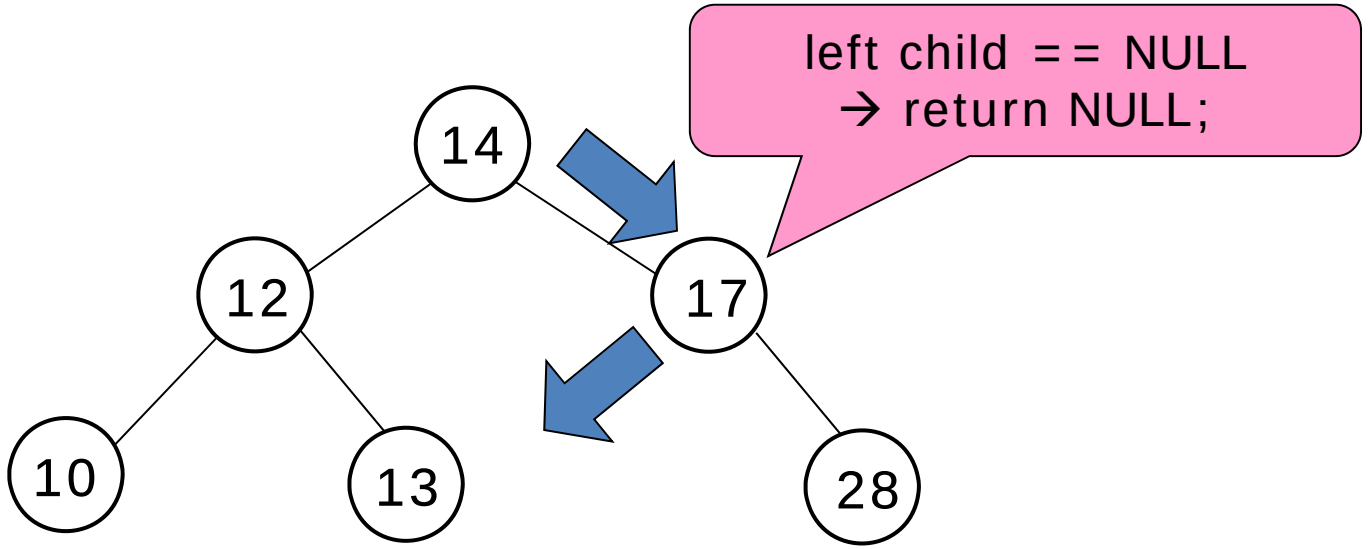
```
element node::search (KEY key )
```



7.5.2 Search

- Given a binary search tree, find a node whose key is k

```
element node::search (KEY key )
```



7.5.2 Search

- Recursive implementation

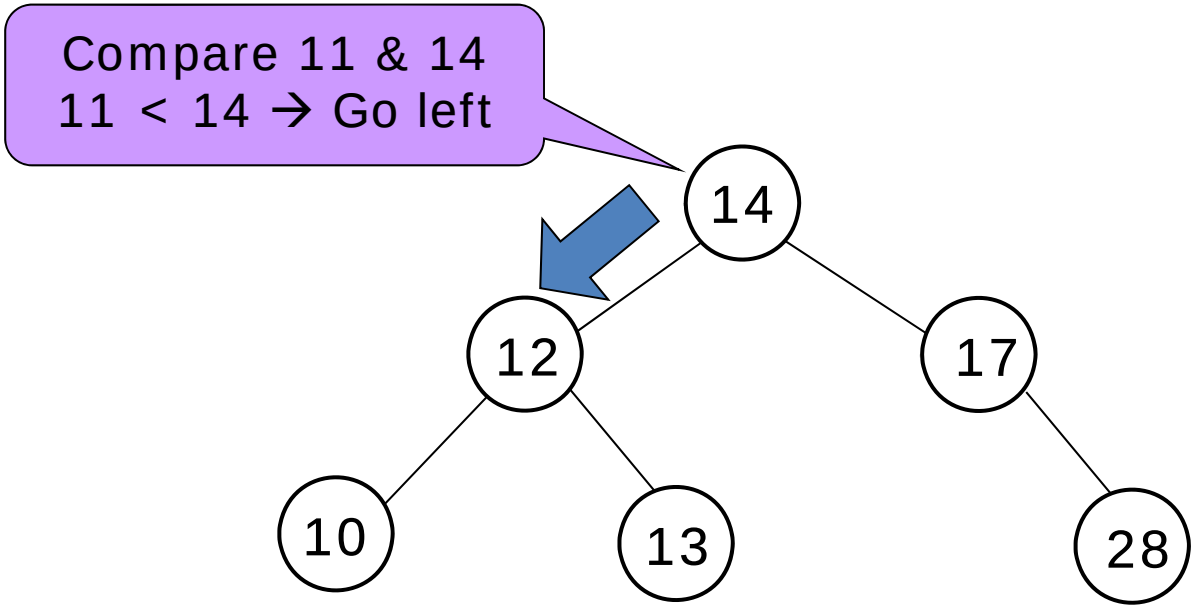
```
void node::search(int ndata)
{
        if (this->key == ndata) {
                printf("found\n");
        }
        else if (this->key < ndata) {
                if (this->rchild != NULL)
                        this->rchild->search(ndata);
                else
                        printf("Not found\n");
        }
        else {
                if (this->lchild != NULL)
                        this->lchild->search(ndata);
                else
                        printf("Not found\n");
        }
}
```

7.5.3 Insert

- Inserting a new node to a binary search tree
  - A newly inserted node is **a leaf node**
  - From the root node of the binary search tree, the key of new node is compared to a leaf node
    - If new key > key of root, then go right
    - If new key < key of root, then go left

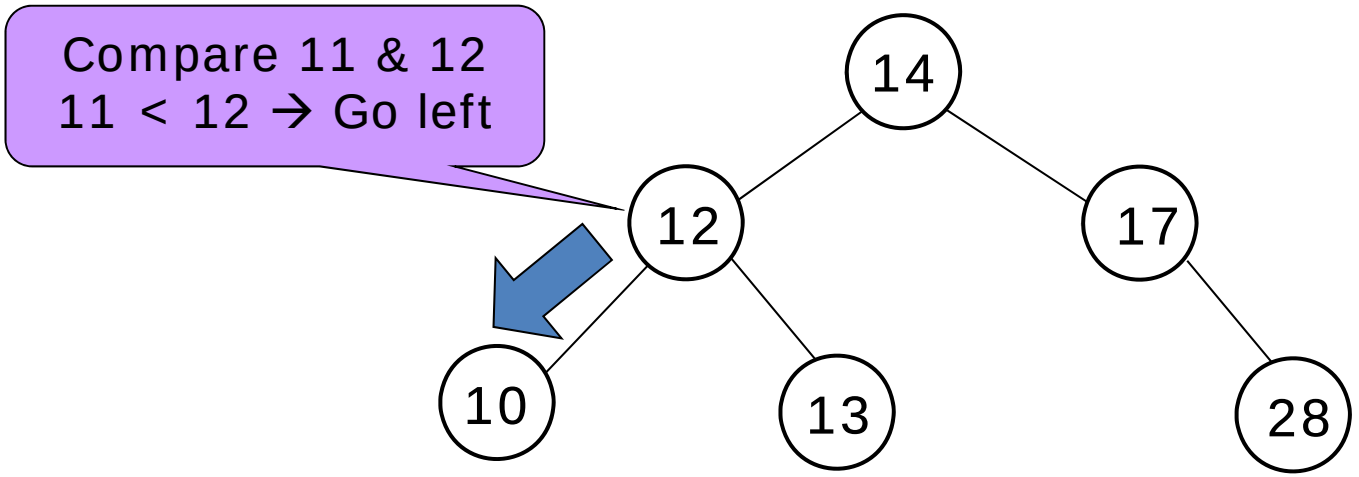
7.5.3 Insert

- Example
  - Insert <11>



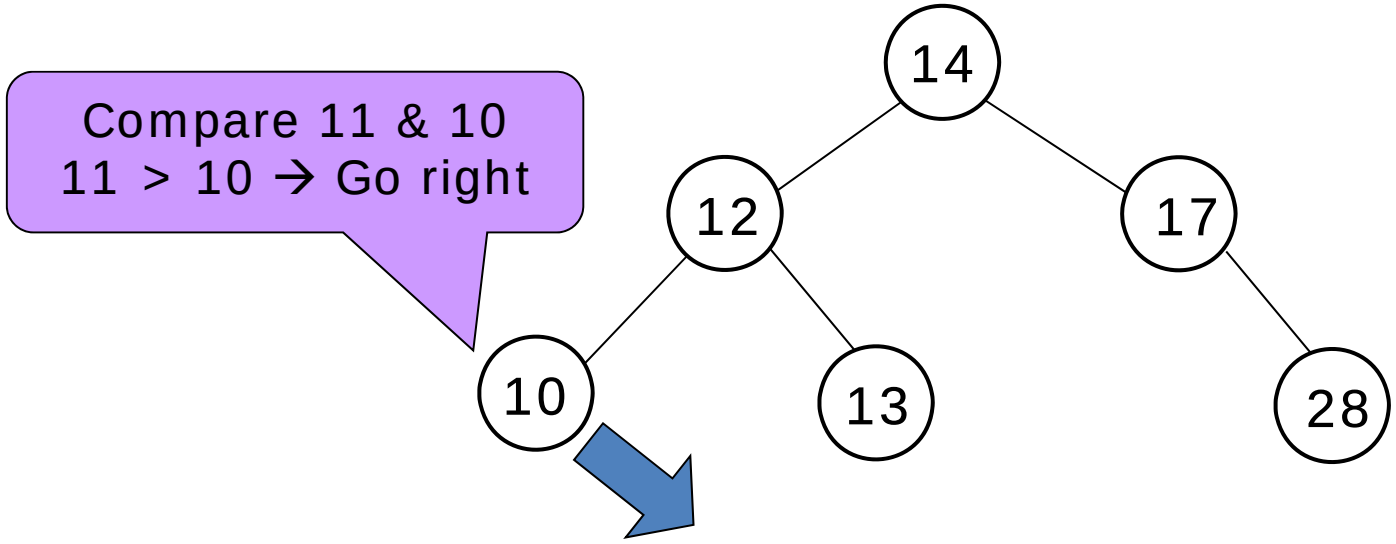
7.5.3 Insert

- Example
  - Insert <11>



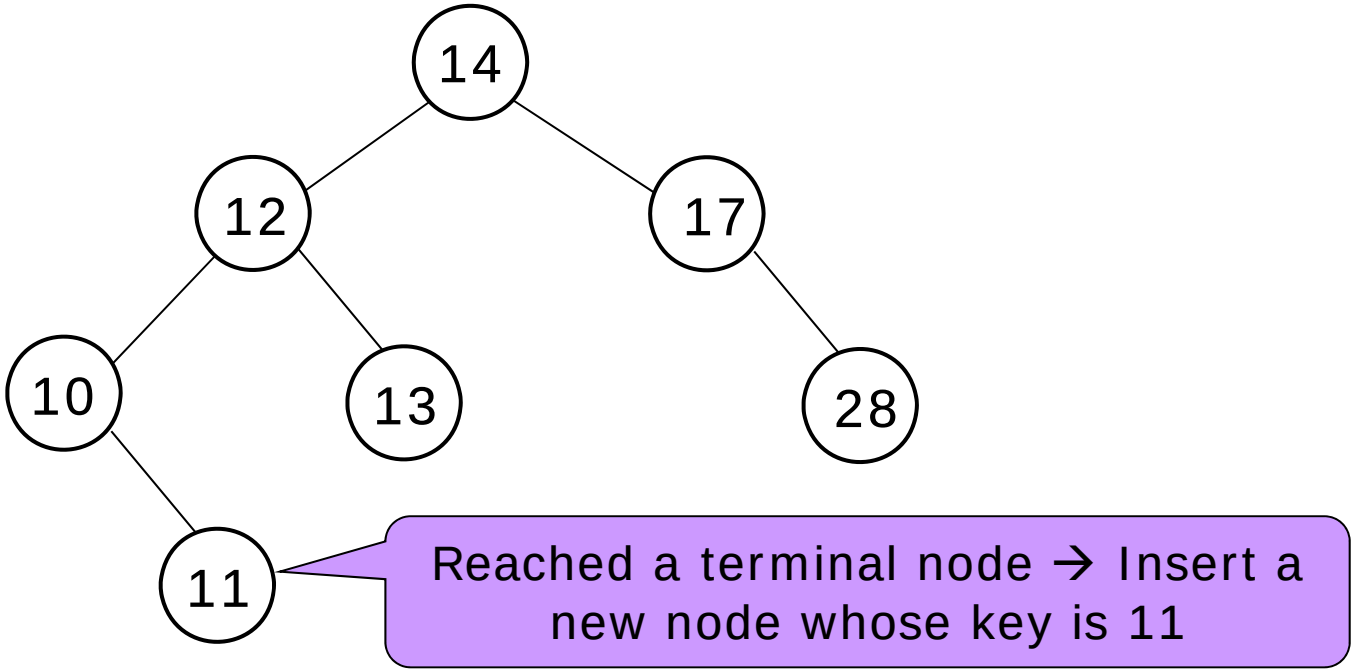
7.5.3 Insert

- Example
  - Insert <11>



7.5.3 Insert

- Example
  - Insert <11>



7.5.3 Insert

- Recursive implementation

```
void node::insert(int ndata)
{
//        degenerate case: root node에 삽입
          if (this->key == -1) {
                    this->key = ndata;
                    return;
          }
//        key보다 크면
          if (ndata > this->key ) {
                    if (this->rchild != NULL)
                              this->rchild->insert(ndata);
                    else {
                              this->rchild = (nptr)malloc(sizeof(node));
                              this->rchild->key = ndata;
                              this->rchild->lchild = this->rchild->rchild = NULL;
                    }
          }
```

7.5.3 Insert

- Recursive implementation

```
//        key보다 작으면
          else if (ndata < this->key ) {
                    if (this->lchild != NULL)
                              this->lchild->insert(ndata);
                    else {
                              this->lchild = (nptr)malloc(sizeof(node));
                              this->lchild->key = ndata;
                              this->lchild->lchild = this->lchild->rchild = NULL;
                    }
          }
//        key와 같으면
          else {
                    printf("no duplicate data\n");
          }
}
```

7.5.4 Delete

- Deleting a node from a binary search tree
  - Which node to delete?
    - Leaf node
    - Internal node with one child node
    - Internal node with two child nodes

7.5.4 Delete

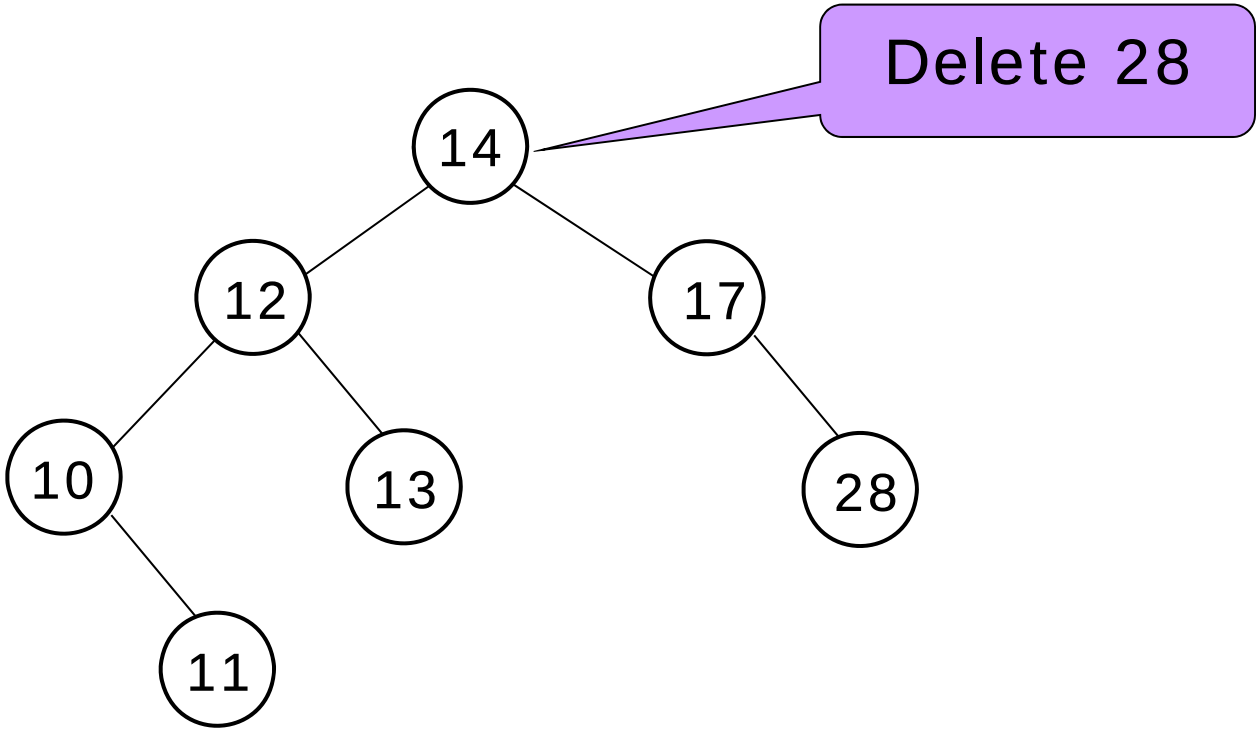
- Deleting leaf nodes

  → Delete the node

7.5.4 Delete

- Deleting leaf nodes

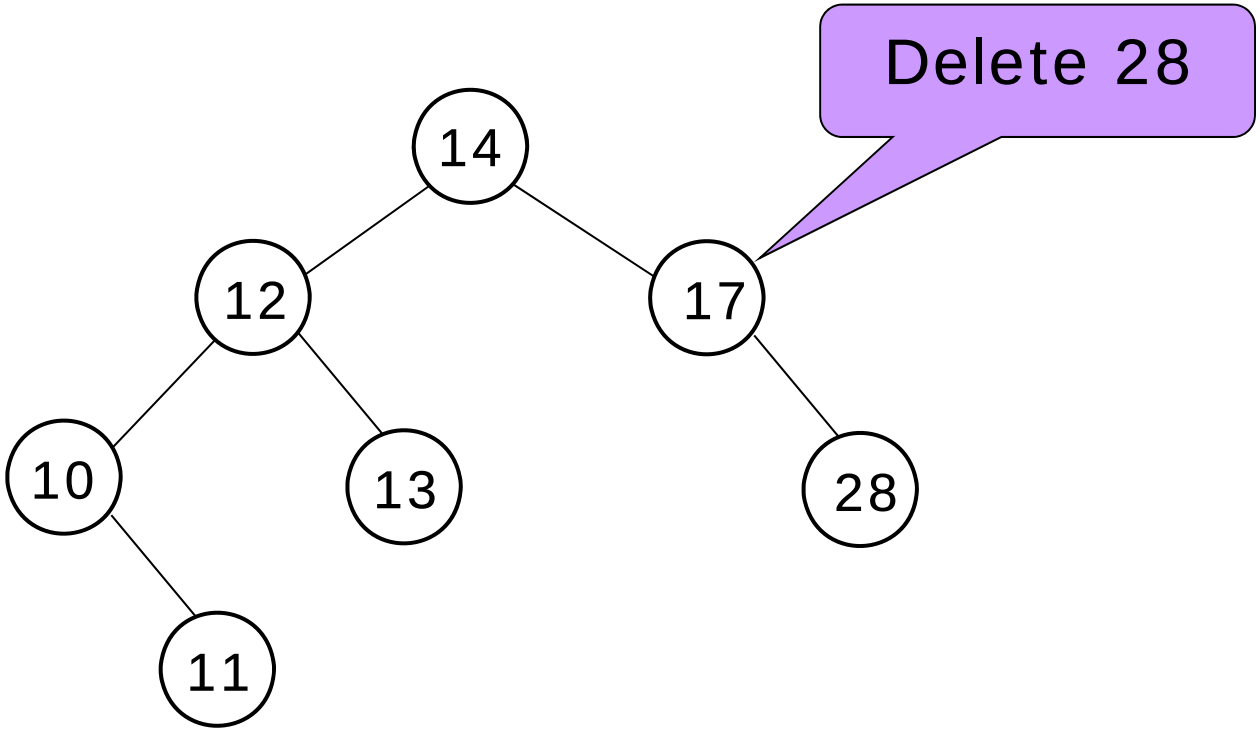
  → Delete the node



7.5.4 Delete

- Deleting leaf nodes

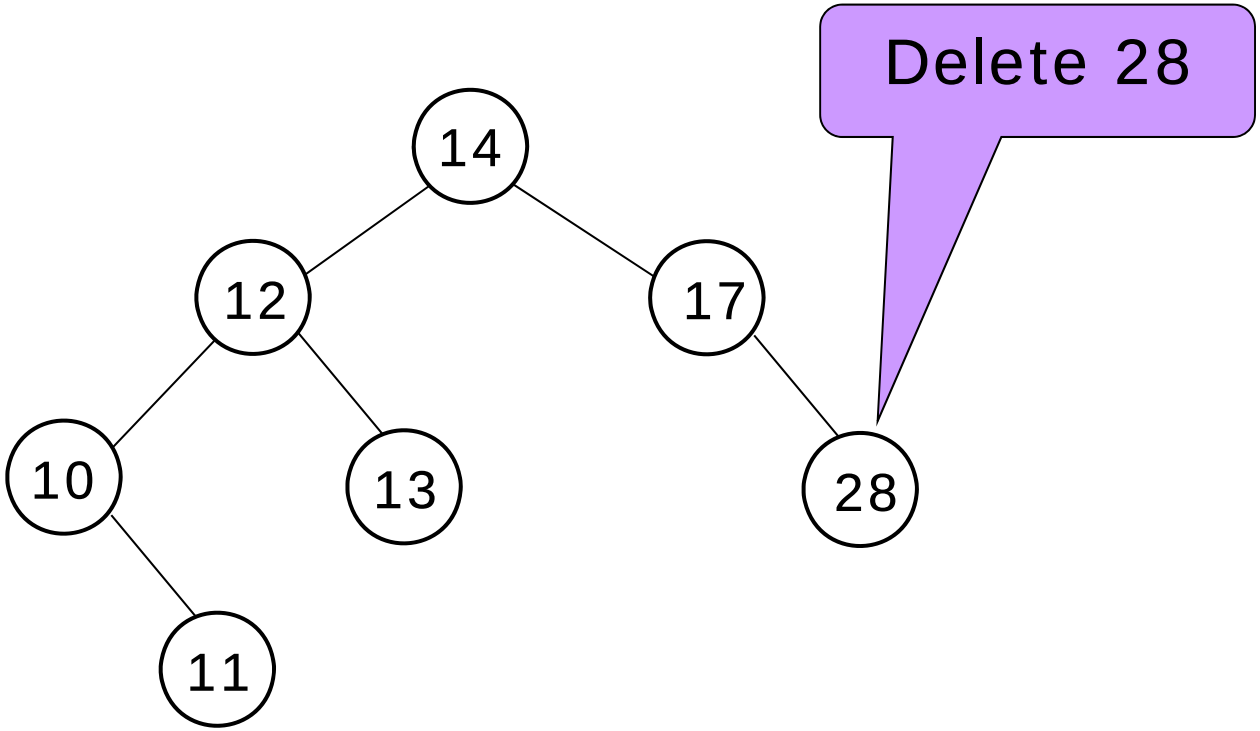
  → Delete the node



7.5.4 Delete

- Deleting leaf nodes

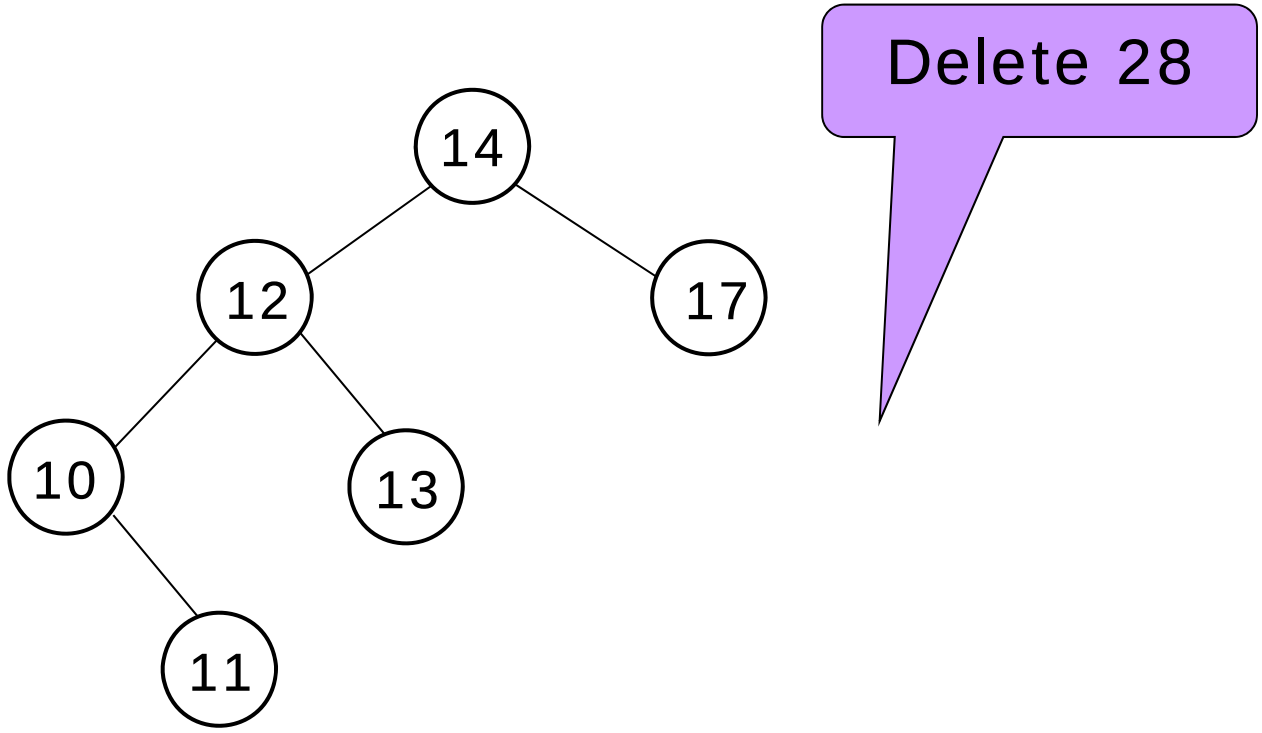
  → Delete the node



7.5.4 Delete

- Deleting leaf nodes

  → Delete the node



7.5.4 Delete

- Deleting internal nodes of one child

  → (1) Delete the node

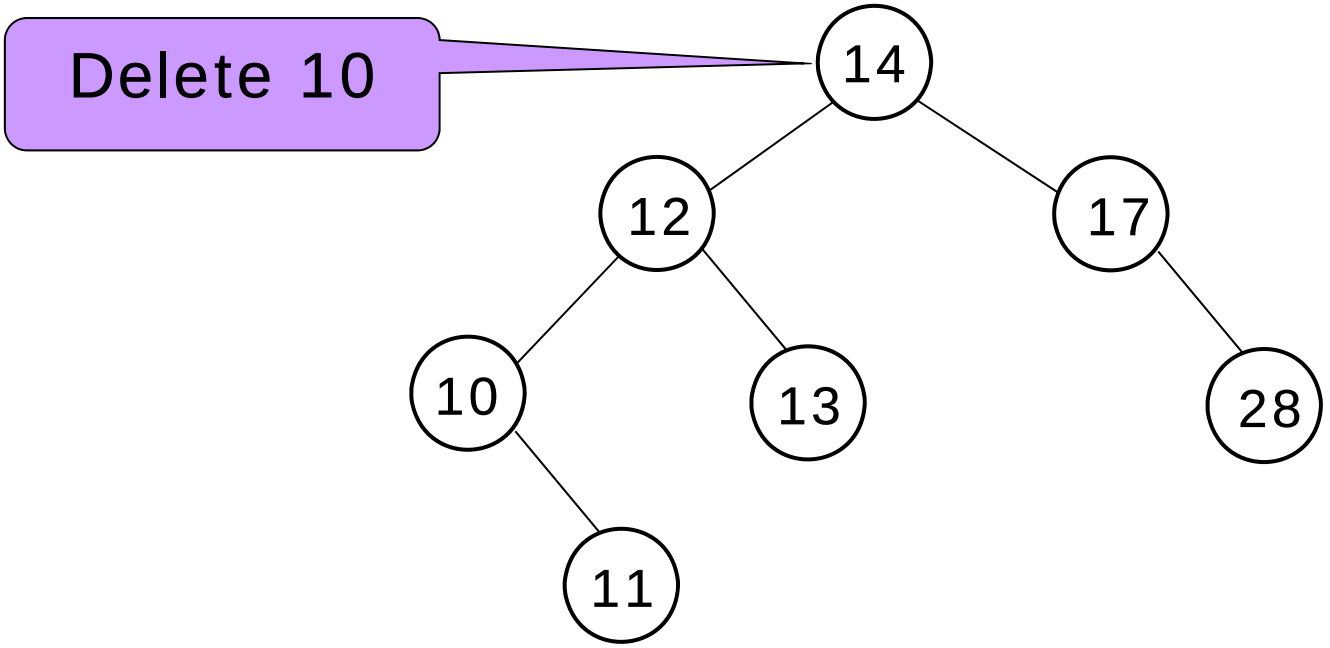
  (2) Make the child take place of the deleted node

7.5.4 Delete

- Deleting internal nodes of one child

  → (1) Delete the node

  (2) Make the child take place of the deleted node

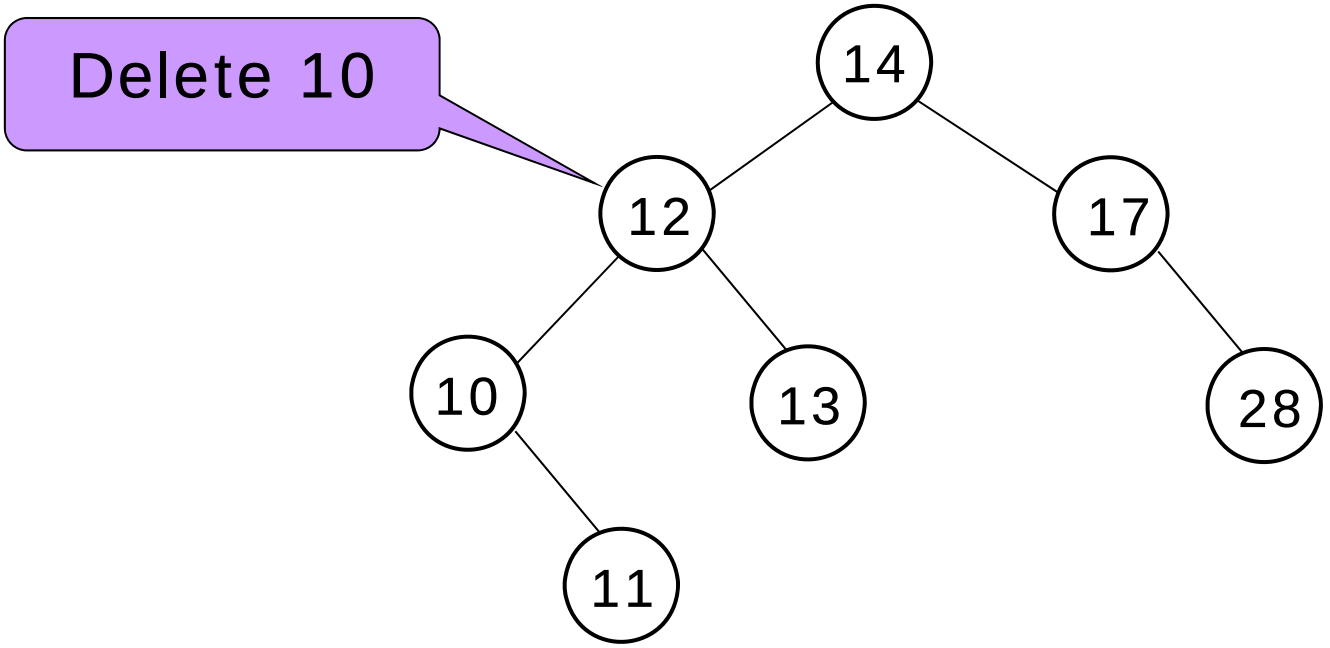


7.5.4 Delete

- Deleting internal nodes of one child

  → (1) Delete the node

  (2) Make the child take place of the deleted node

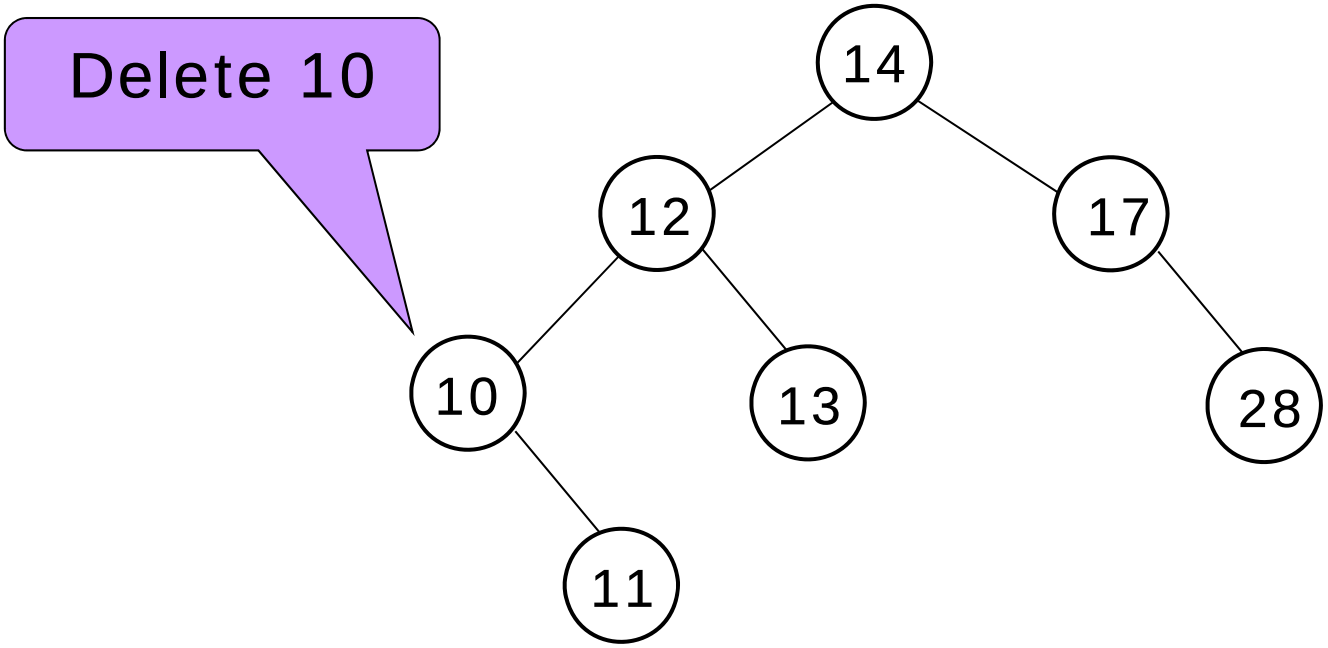


7.5.4 Delete

- Deleting internal nodes of one child

  → (1) Delete the node

  (2) Make the child take place of the deleted node

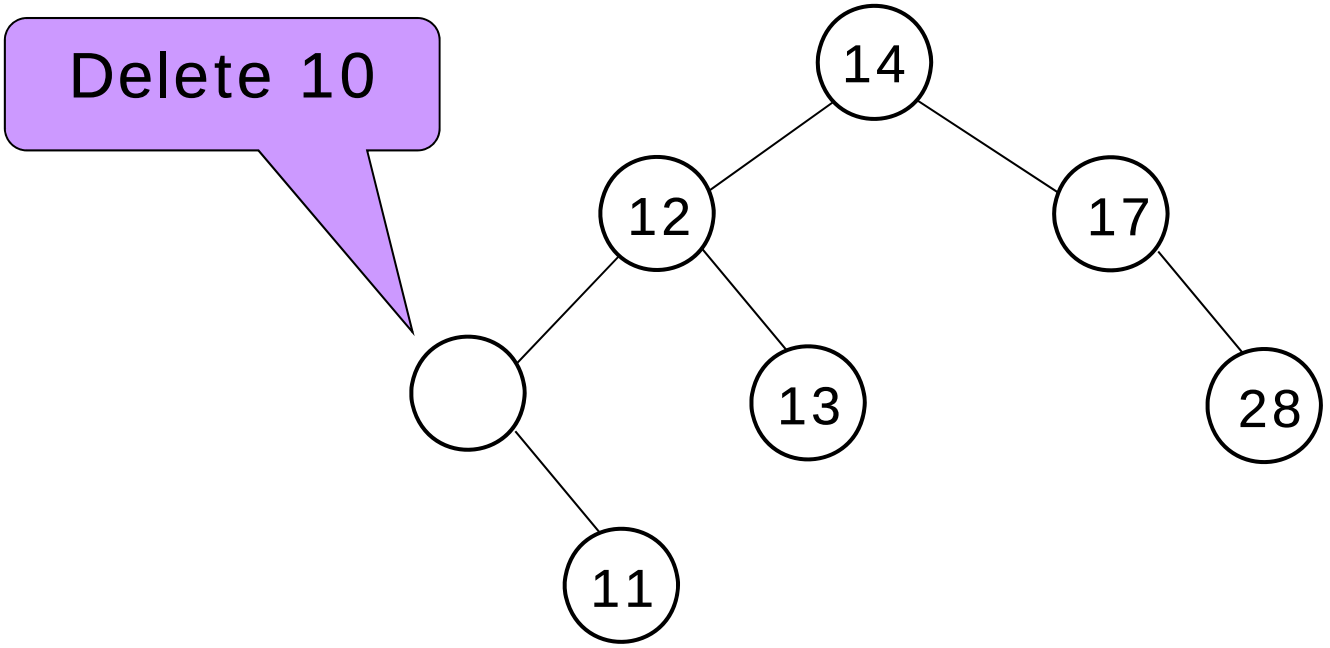


7.5.4 Delete

- Deleting internal nodes of one child

  → (1) Delete the node

  (2) Make the child take place of the deleted node

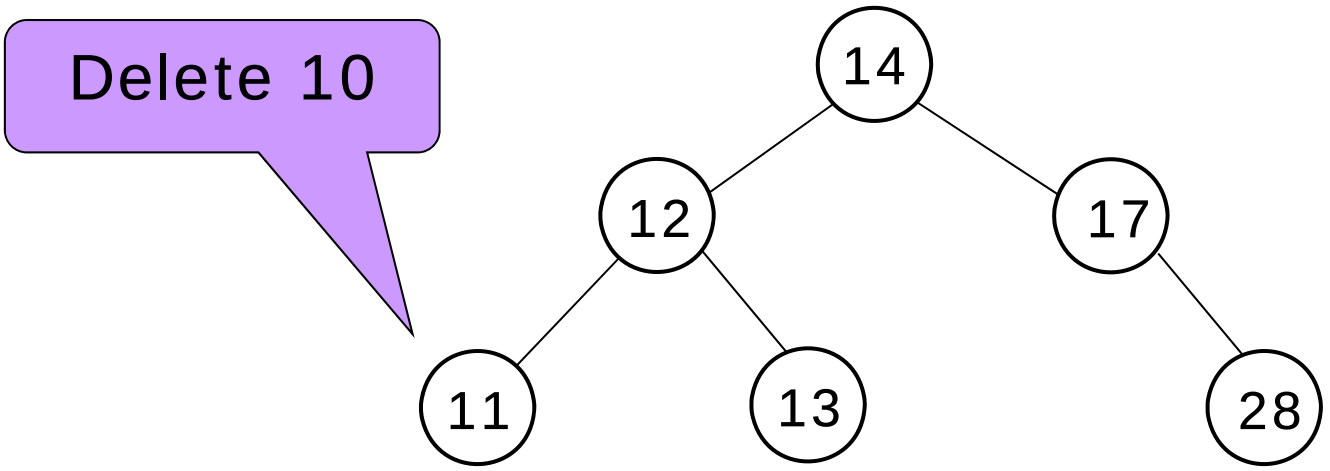


7.5.4 Delete

- Deleting internal nodes of one child

  → (1) Delete the node

  (2) Make the child take place of the deleted node

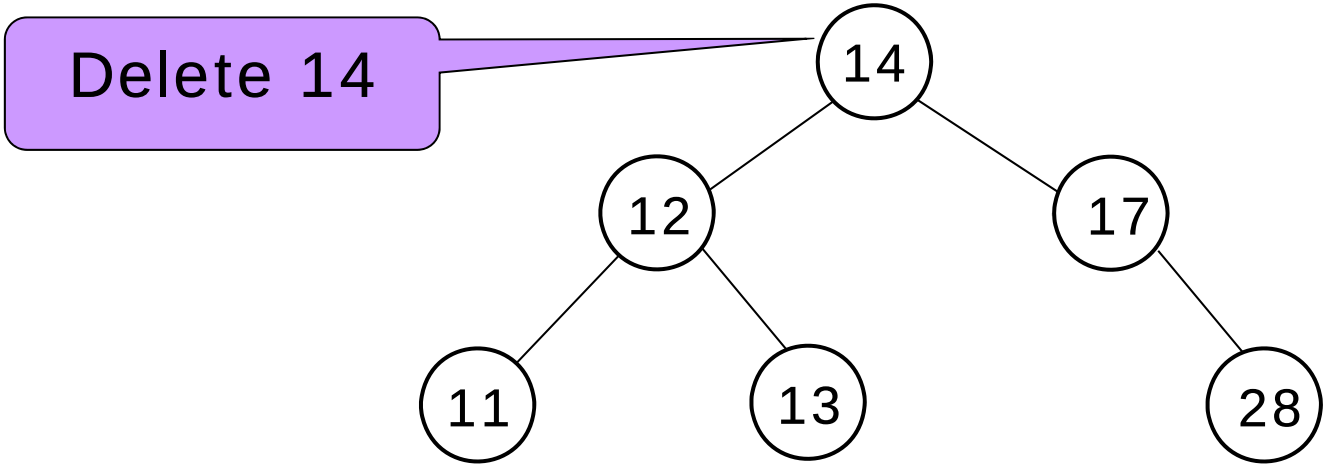


7.5.4 Delete

- Deleting internal nodes with two childs
  → (1) Delete the node
  (2) Move the maximum of its left subtree
  (or the minimum of its right subtree)
  to the node

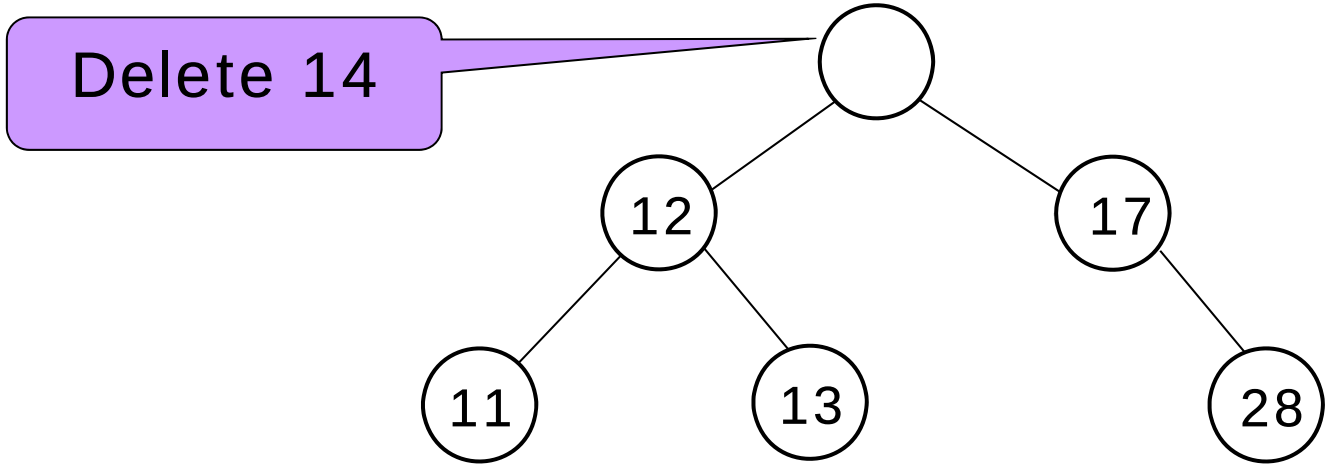
7.5.4 Delete

- Deleting internal nodes with two childs
  - → (1) Delete the node
    (2) Move the maximum of its left subtree (or the minimum of its right subtree) to the node



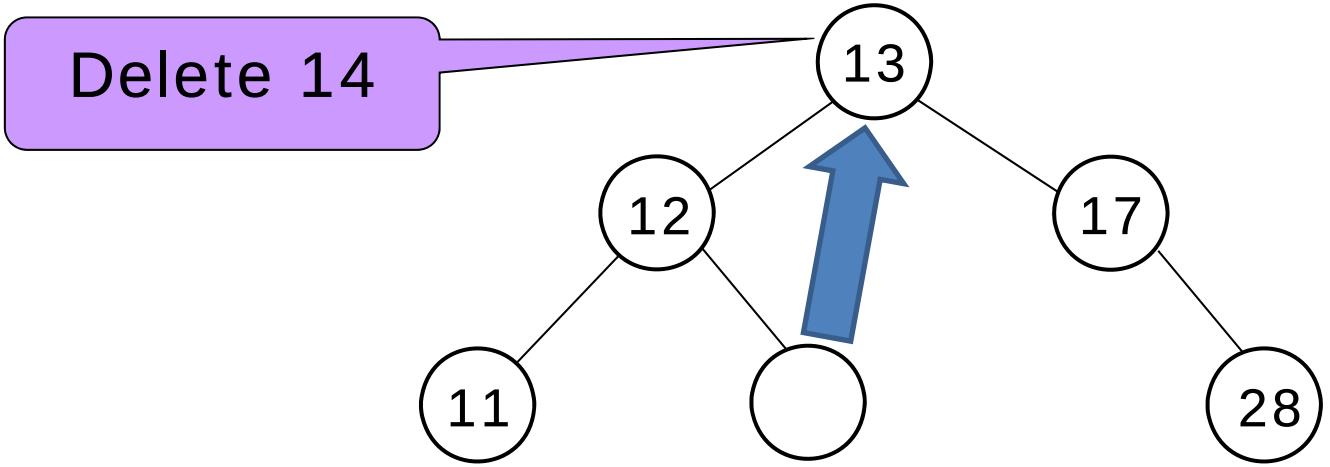
7.5.4 Delete

- Deleting internal nodes with two childs
  - → (1) Delete the node
    (2) Move the maximum of its left subtree (or the minimum of its right subtree) to the node



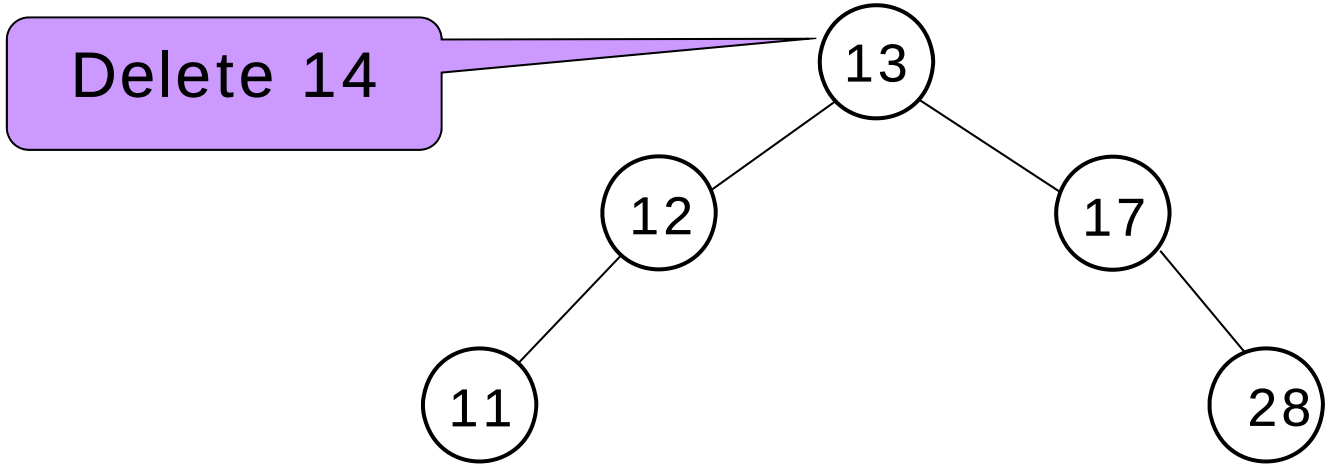
7.5.4 Delete

- Deleting internal nodes with two childs
  - → (1) Delete the node
    (2) Move the maximum of its left subtree (or the minimum of its right subtree) to the node



7.5.4 Delete

- Deleting internal nodes with two childs
  - → (1) Delete the node
    (2) Move the maximum of its left subtree (or the minimum of its right subtree) to the node



7.5.4 Delete

- Recursive implementation

```
int node::remove(int ndata)
{
        if (this->key == ndata) {
                printf("Removing %d\n", ndata);
        //      (1) 두 child가 다 NULL → leaf node이면
                if (this->lchild == NULL && this->rchild == NULL)
                        return 1;
        //      (2) left child만 NULL이면
                if (this->lchild == NULL && this->rchild != NULL) {
                        this->key = this->rchild->key;
                        this->lchild = this->rchild->lchild;
                        this->rchild = this->rchild->rchild;
                        return 0;
                }
        //      (3) right child만 NULL이면
                if (this->lchild != NULL && this->rchild == NULL) {
                        this->key = this->lchild->key;
                        this->rchild = this->lchild->rchild;
                        this->lchild = this->lchild->lchild;
                        return 0;
                }
```

7.5.4 Delete

- Recursive implementation

```
        //      (4) 두 child가 다 NULL이 아닌 경우
                if (this->lchild != NULL && this->rchild != NULL) {
                        this->key = this->lchild->get_max();
                        this->lchild->remove(this->key );
                        return 0;
                }
        }
```

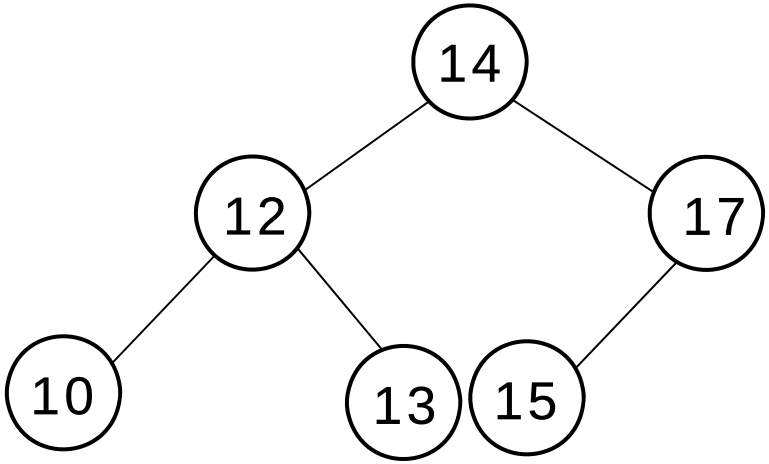
7.5.4 Delete

- Recursive implementation

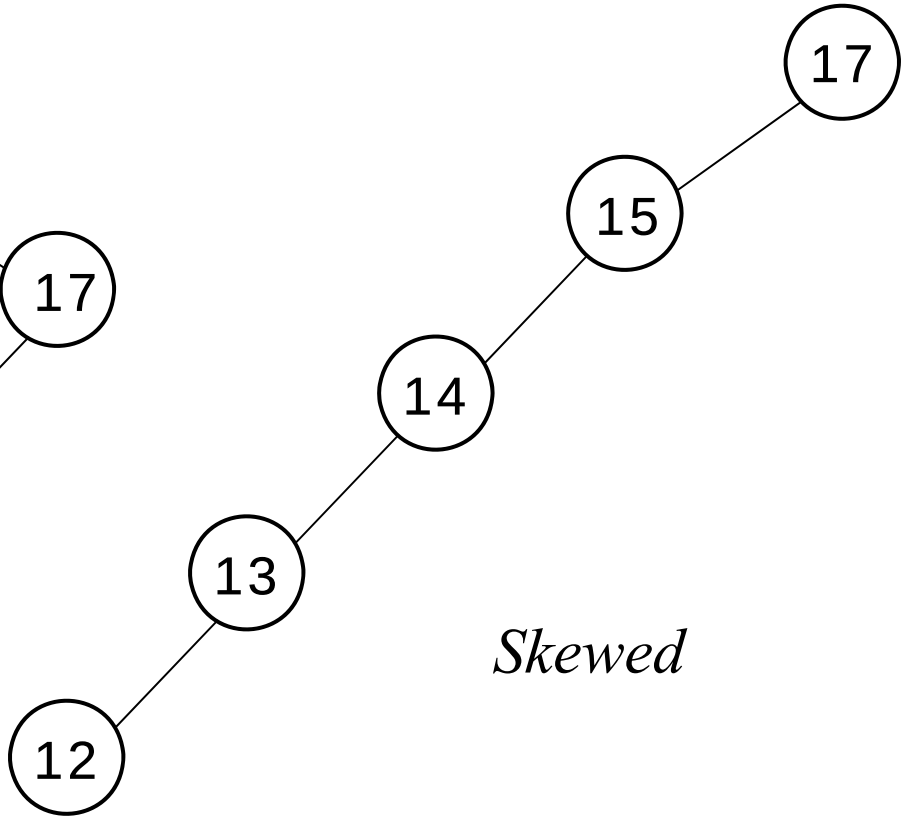
```
        else if (this->key < ndata) {
                if (this->rchild != NULL) {
                        if (this->rchild->remove(ndata))
                                this->rchild = NULL;
                }
                else {
                        printf("Not found %d in removing\n", ndata);
                }
                return 0;
        }
        else {
                if (this->lchild != NULL) {
                        if (this->lchild->remove(ndata))
                                this->lchild = NULL;
                }
                else {
                        printf("Not found %d in removing\n", ndata);
                }
                return 0;
        }
}
```

7.5.5 Time complexity

- Balanced (best) VS Skewed (worst)



Balanced



Skewed

7.5.5 Time complexity

- Data structures for efficient search

Data structure		**Insert**	**Delete**	**Search**	**Get max (Pop)**	**Remove max (Top)**
Array	Unsorted	O(1)	O(n)	O(n)	O(n)	O(n)
	Sorted	O(n)	O(n)	O(log n)	O(1)	O(n)
Linked list	Unsorted	O(n)	O(n)	O(n)	O(n)	O(n)
	Sorted	O(n)	O(n)	O(n)	O(1)/O(n)	O(1)/O(n)
Binary search tree	*BC*	***O(log n)***	***O(log n)***	***O(log n)***	***O(log n)***	***O(log n)***
	WC	*O(n)*	*O(n)*	*O(n)*	*O(n)*	*O(n)*
Heap						
Hash table						

7.5.6 Advanced topics

- The key issue in BST
  - How to keep the balance?
  - Ex) Insert 1, 2, 3, 4, 5, 6, 7, 8
  - Ex) Insert 5, 3, 7, 2, 6, 1, 8, 4

7.5.6 Advanced topics

- The key issue in BST
  - Automatically balancing trees
    - AVL tree
    - 2-3 tree
    - Red-black tree
    - Spray tree
    - B or B+ tree
    - ……