자료구조

Chap 7. Tree

2017년 2학기

컴퓨터과학과
민 경 하

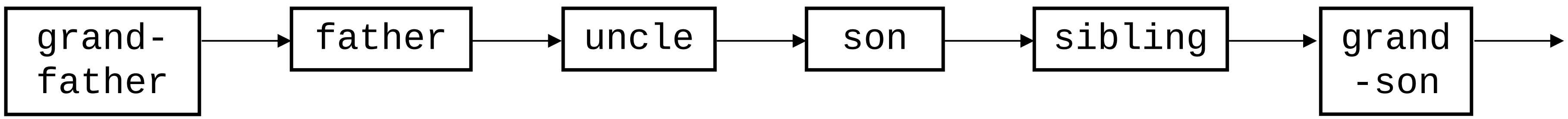
Contents

7.1 Introduction

- What are the common points of array, stack, queue and linked list?
  - Linear data structure
  - prev → curr → next
  - Each element is mapped with index

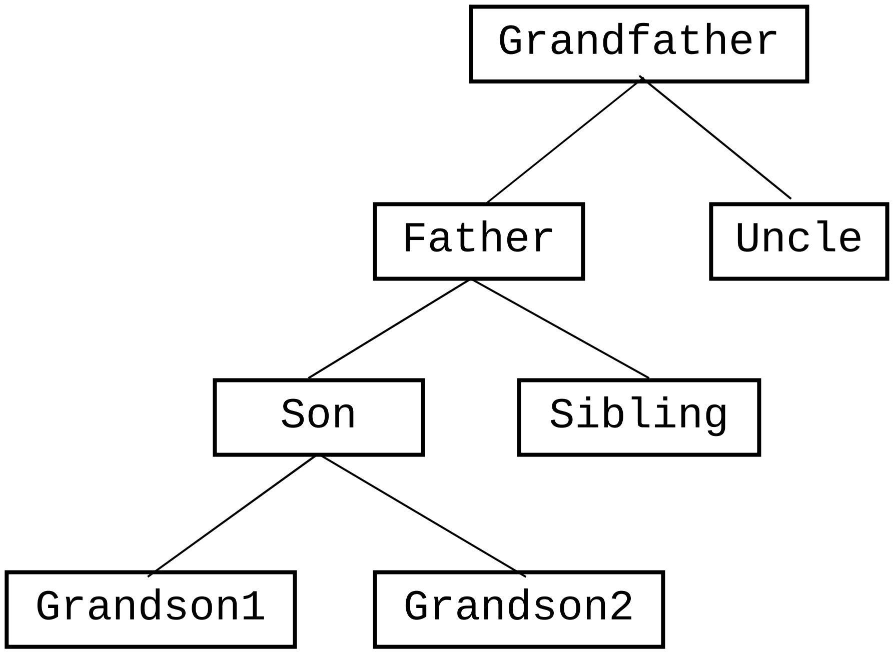
7.1 Introduction

- Limitations of linear data structure?
  - Representation of “family record (족보)”
    - grandfather
    - father, uncle
    - son, sibling
    - grandson1, grandson2



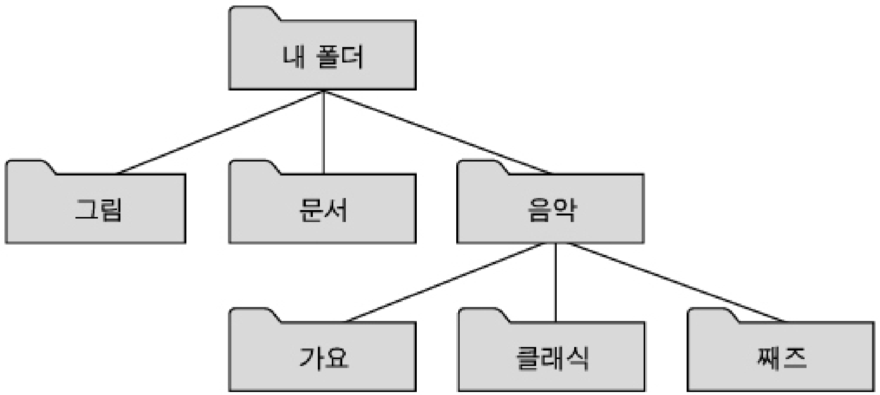
7.1 Introduction

- Representation of family record



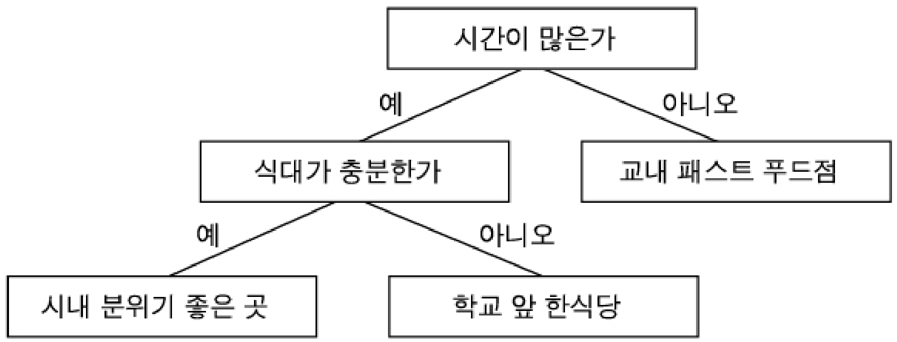
7.1 Introduction

- Similar data structures:
  - File organization



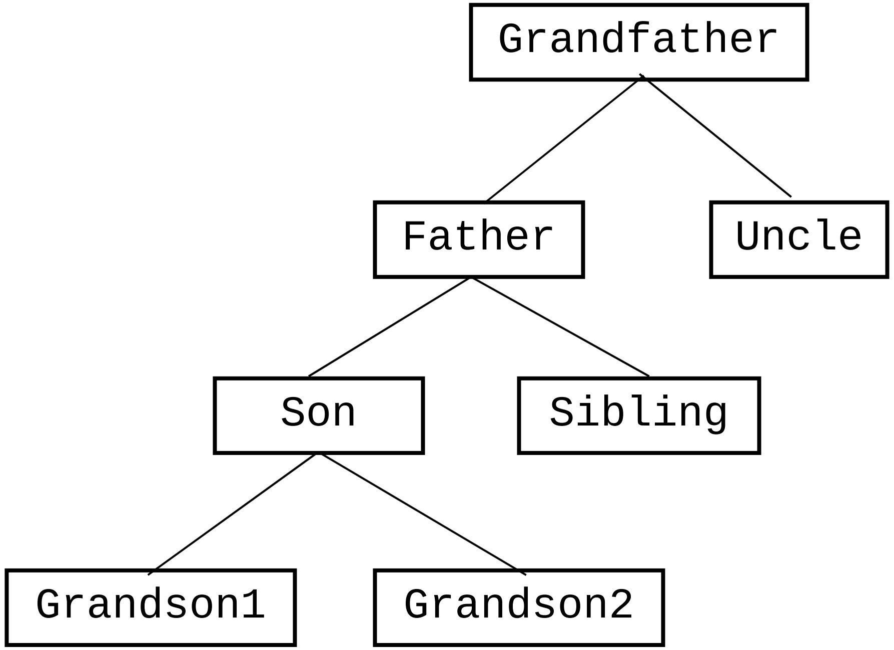
7.1 Introduction

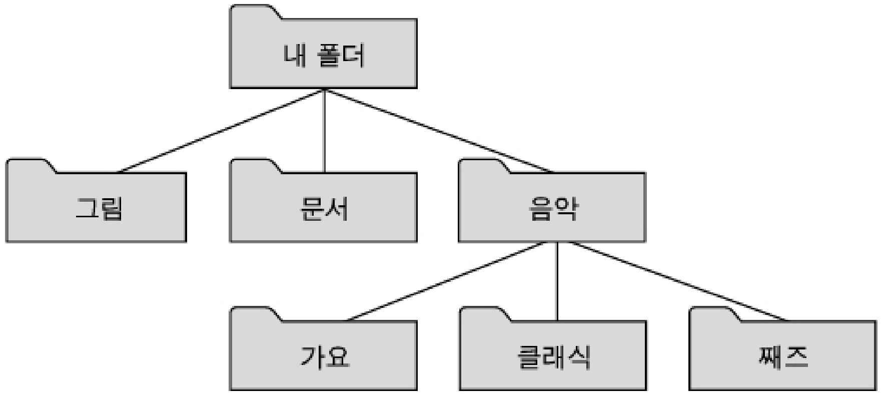
- Similar data structures:
  - Decision making

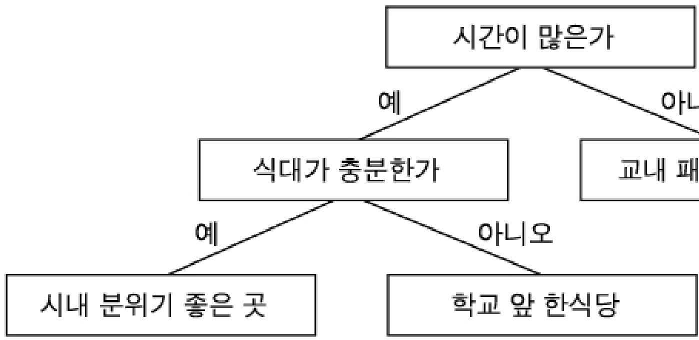


7.1 Introduction

- Hierarchical data structure

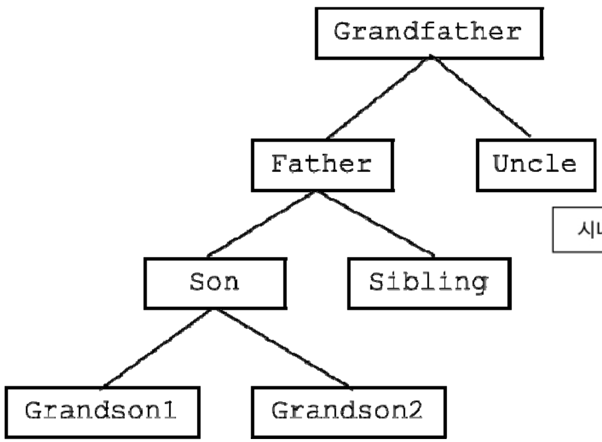


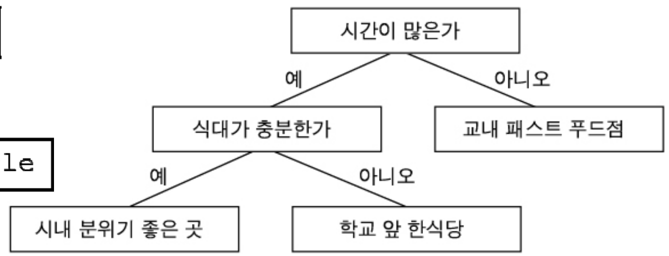


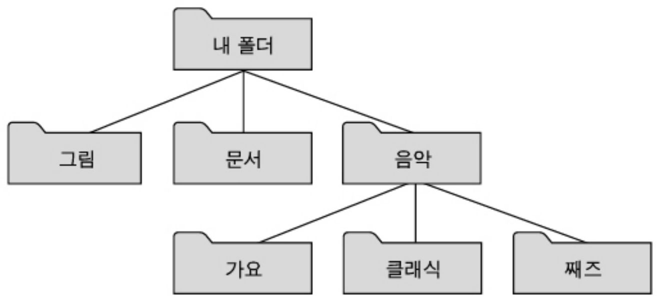


7.1 Introduction

- What is common to these structures?
  - (1) Originated from one source
  - (2) One node is propagated into several nodes
  - (3) No cycle path

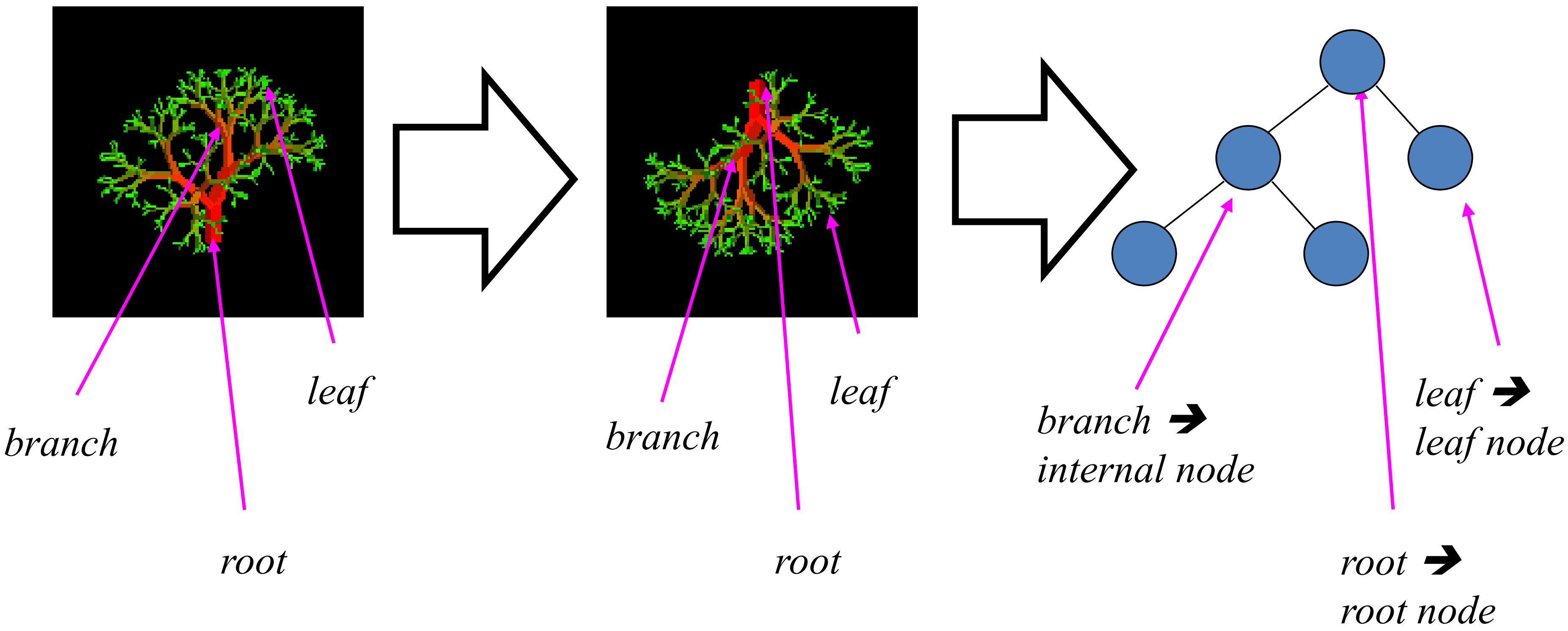






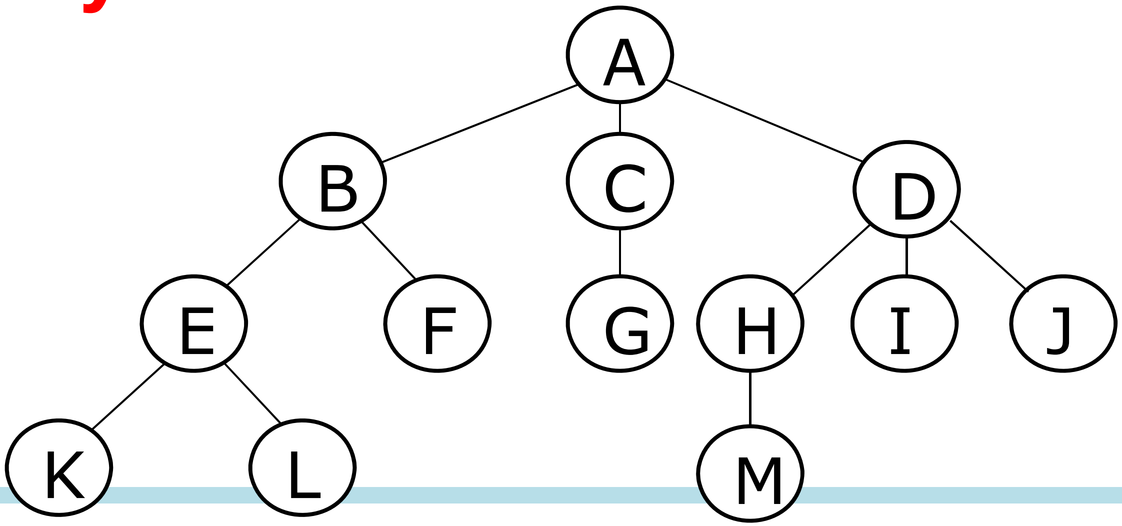
7.1 Introduction

- Hierarchical data structure → tree



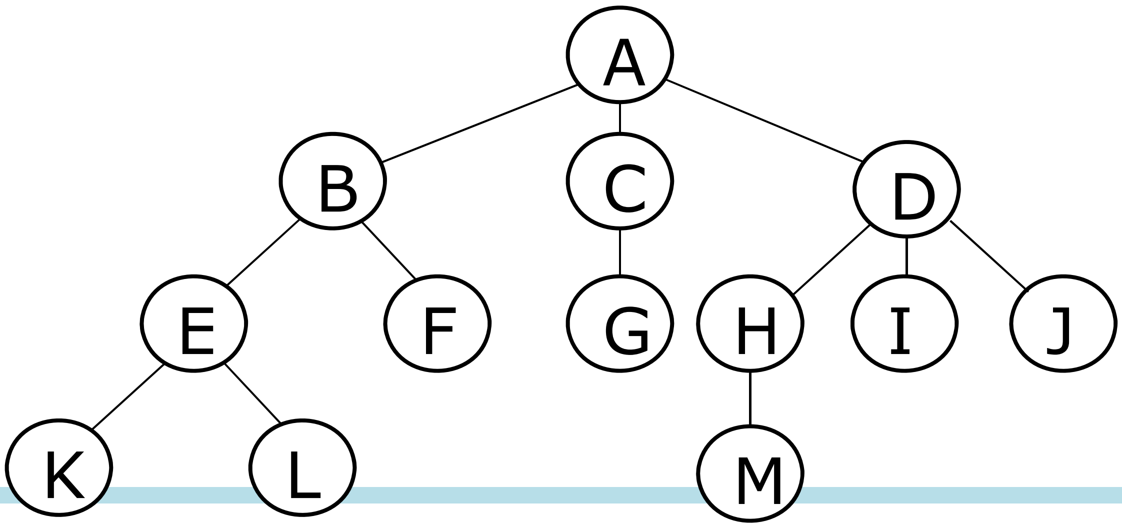
7.2 Basic concepts

- Definition of a tree
  (1) There is a special designated node call the **root**
  (2) Every pairs of connected nodes are in **parent-child** relationship
  (3) There is **no cycle** in the nodes



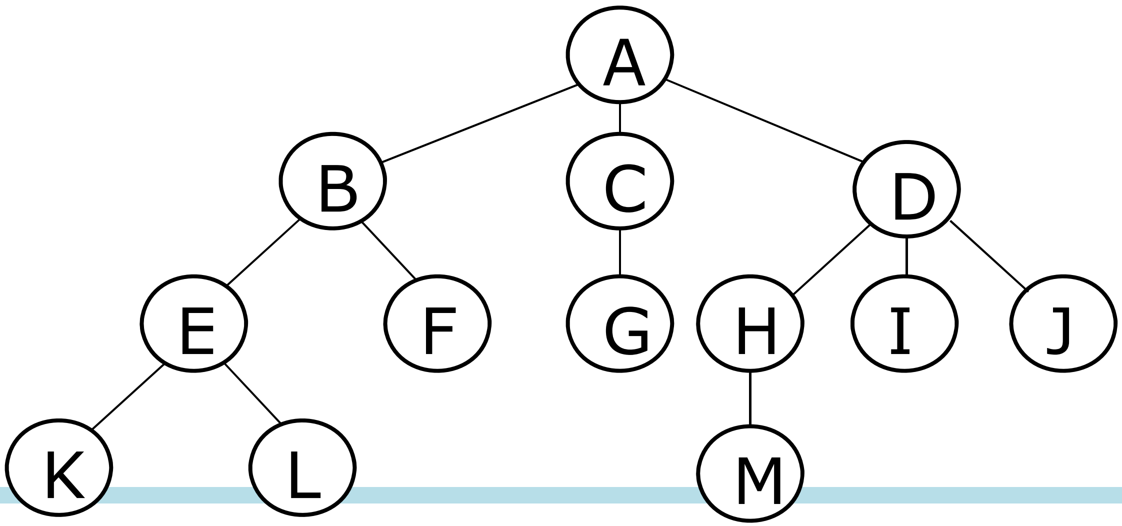
7.2 Basic concepts

- Terms (1)
  - Node (or vertex)
  - Edge



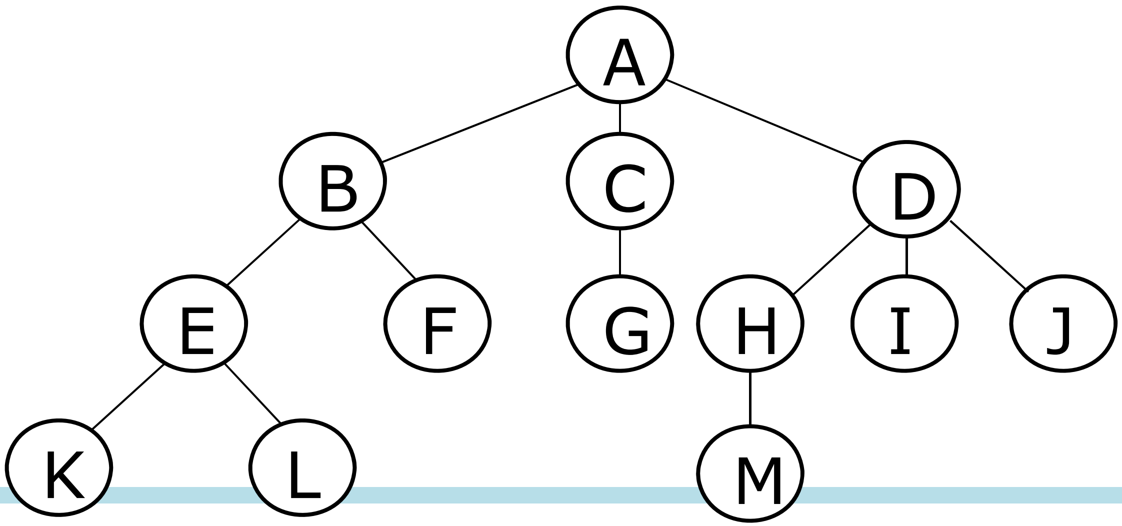
7.2 Basic concepts

- Terms (2)
  - Root node
  - Leaf node
  - Internal node



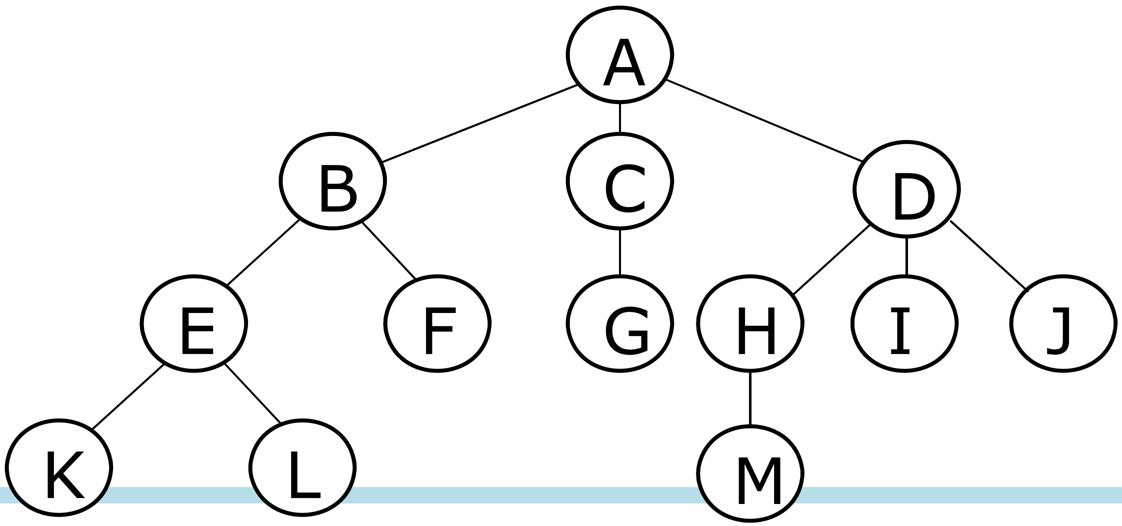
7.2 Basic concepts

- Terms (3)
  - Parent node
  - Child node
  - Sibling node



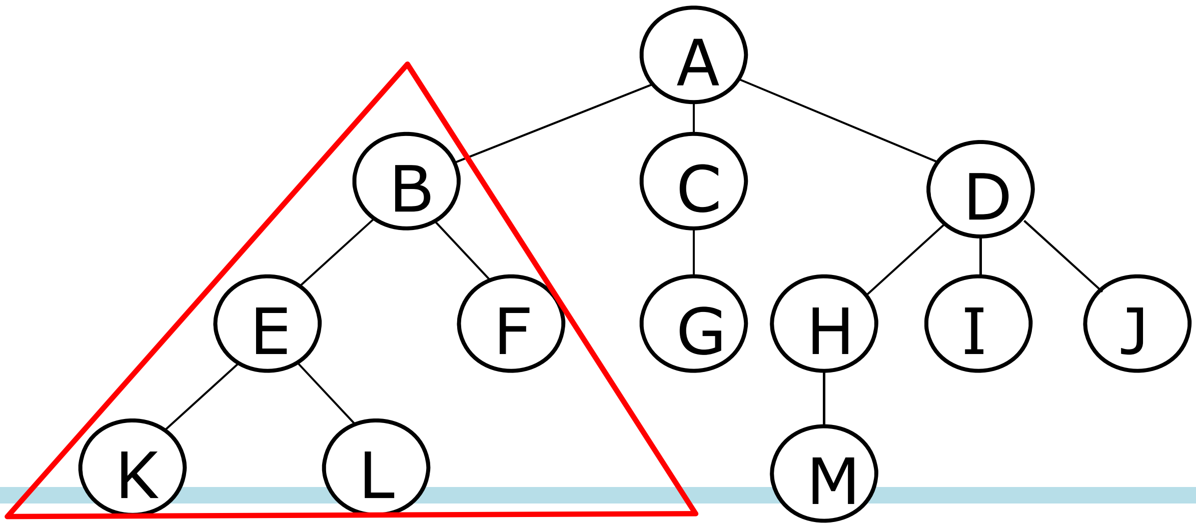
7.2 Basic concepts

- Terms (4)
  - Ancestor node
  - Descendent node



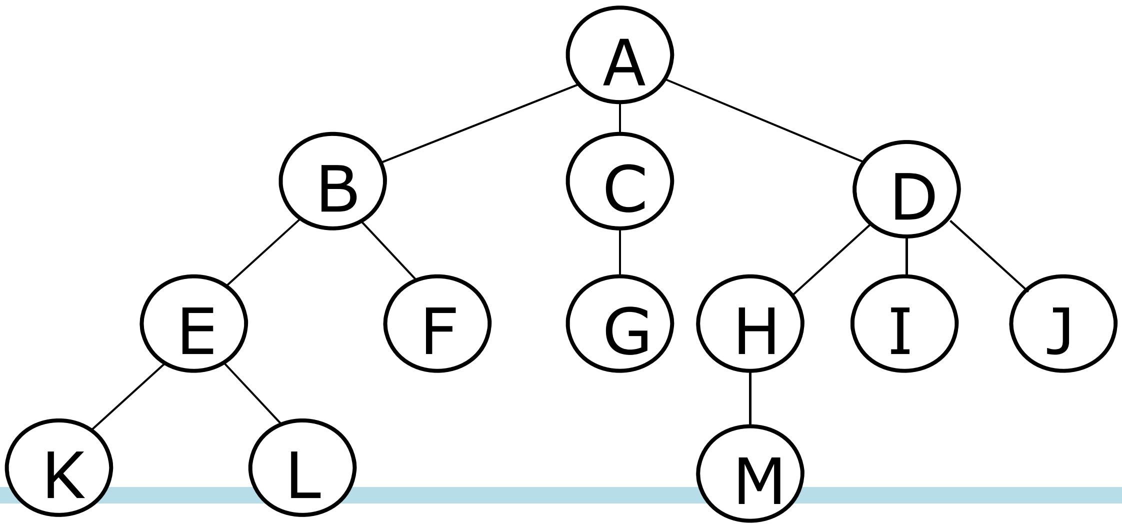
7.2 Basic concepts

- Terms (5)
  - subtree



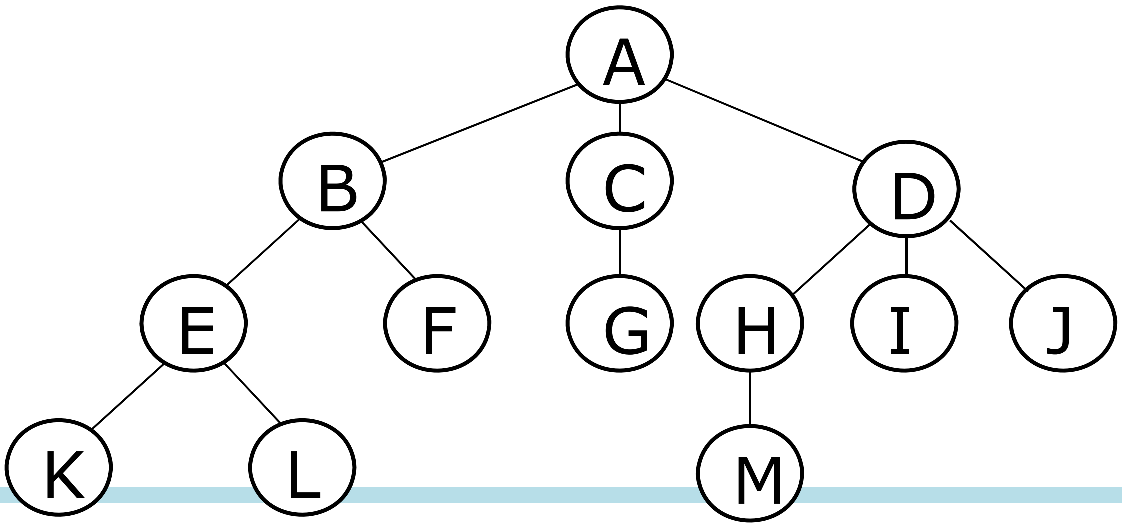
7.2 Basic concepts

- Terms (6)
  - Degree of a node
  - Degree of a tree
    - Binary tree
    - Ternary tree
    - K-ary tree



7.2 Basic concepts

- Terms (7)
  - Depth of a node
  - Depth (height) of a tree
    - Depth of a root = 1
  - Width of a tree



7.2 Basic concepts

- Data structure of a tree
  - Node
    - Data
    - No. of child nodes
    - Pointers to the child nodes
  - Pointer-based structure

```
typedef class node *nptr;
class node {
    data_type data;
    int n_childs;
    nptr *childs;
};
```

7.2 Basic concepts

- Organizations of tree

Tree (7.1 & 7.2)
- Definition
- Basic concept
- Traversal (BFS, DFS)

Binary tree (7.3 & 7.4)
- Definition
- Basic properties
- Basic operations
- Traversal (inorder, preorder, postorder)

Binary search tree (7.5)
- Definition
- Search
- Insert/delete

Heap (7.6)
- Definition
- Insert
- Delete

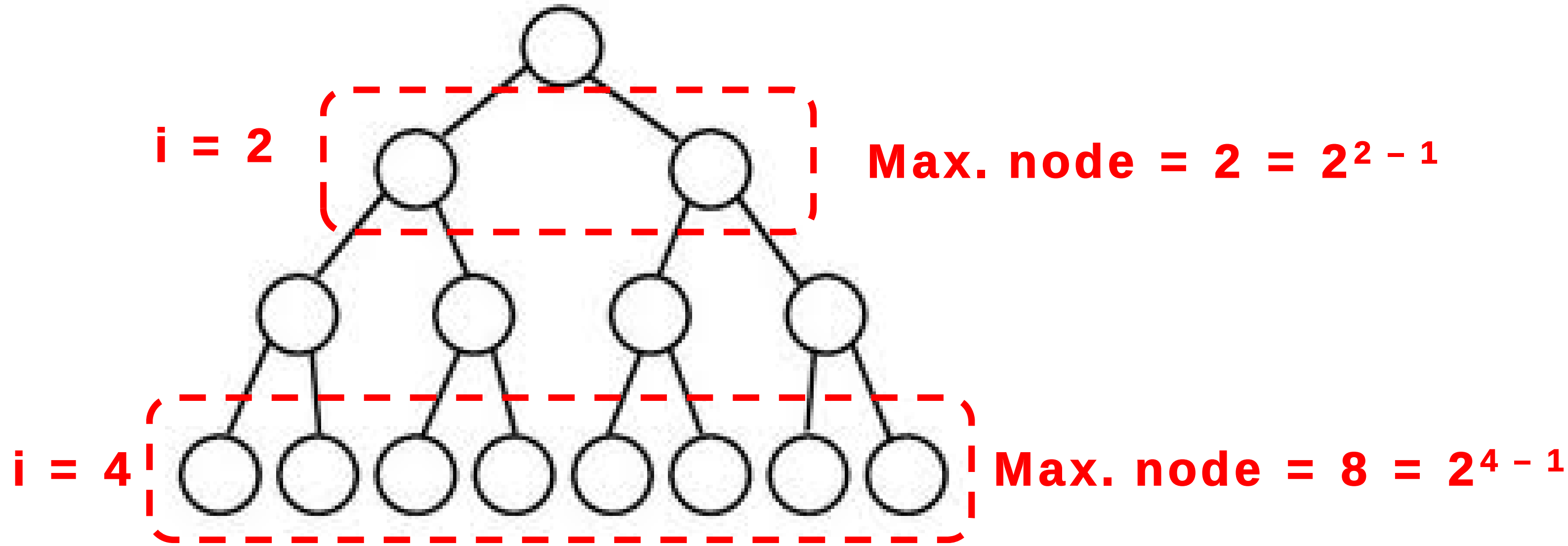
7.3 Binary tree

- Definition
  - A tree whose degree is 2
  - The maximum degree of its nodes is 2

```
typedef class node *nptr;
class node {
    data_type data;
    nptr lchild, rchild;
};
```

7.3 Binary tree

- Properties of a binary tree
  - (1) The maximum nodes in i-th level is 2^{i-1}.
    - Proof) Use mathematical induction

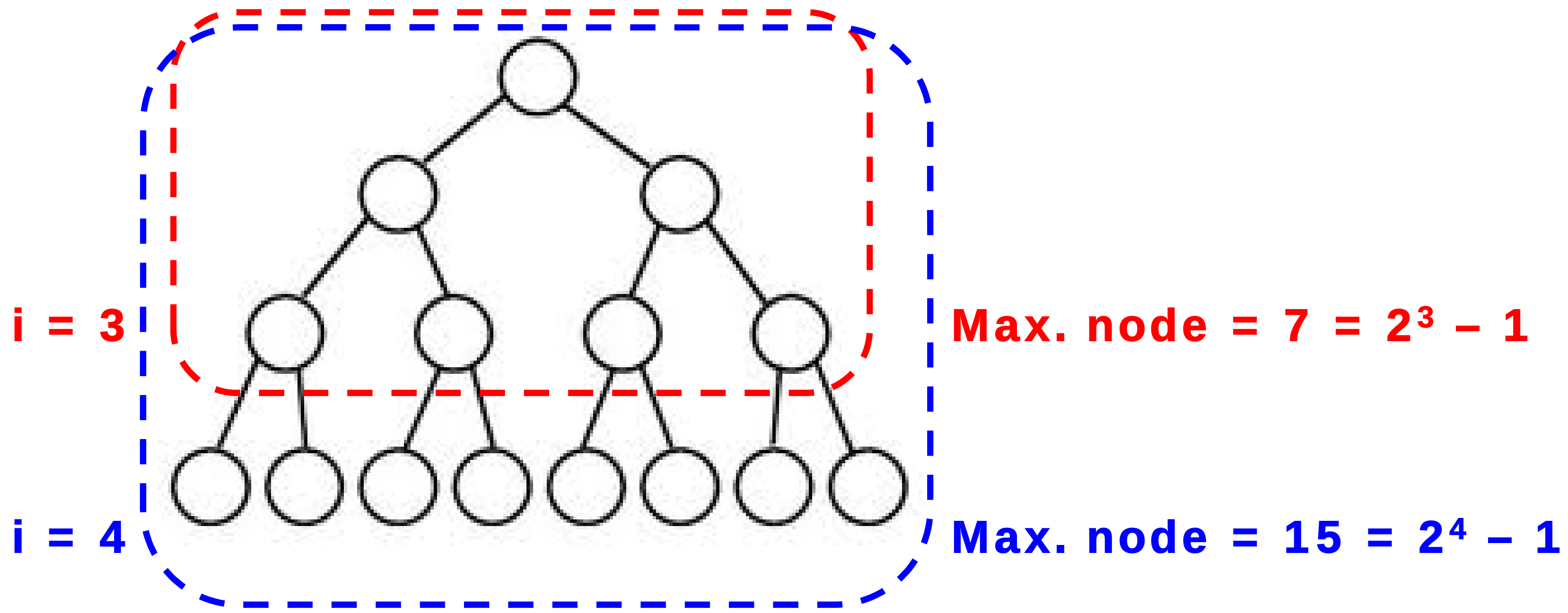


7.3 Binary tree

- Properties of a binary tree

  (2) The maximum nodes of a binary tree of depth k is $2^k - 1$.

  Proof) Use mathematical induction



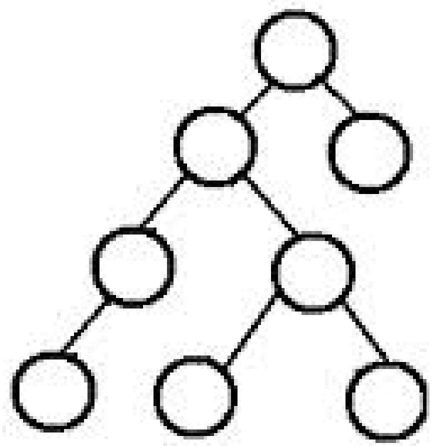
7.3 Binary tree

- Properties of a binary tree

(3) For any nonempty binary tree, if n_0 is the number of leaf nodes and n_2 the number of nodes of degree 2, then $n_0 = n_2 + 1$.

proof) (i) $n_0 + n_1 + n_2 = n$

(ii) $2n_2 + n_1 + 1 = n$



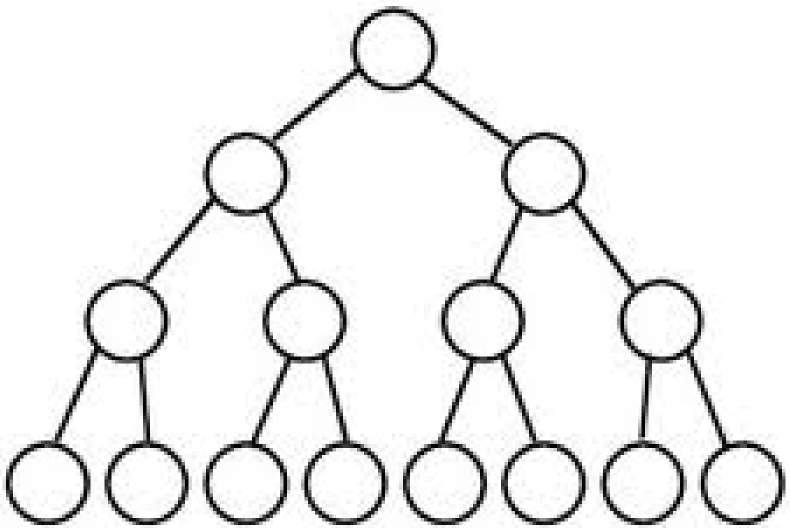
$\mathbf{n_0 = 4}$
$\mathbf{n_2 = 3}$
$\mathbf{n_0 = n_2 + 1}$

7.3 Binary tree

- Special binary trees

  (1) Full binary tree

  A full binary tree of depth k is a binary tree of depth k having $2^k - 1$ nodes, $k \geq 1$.

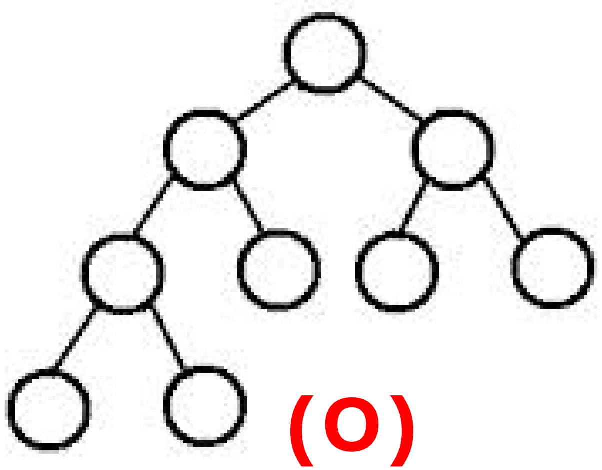


7.3 Binary tree

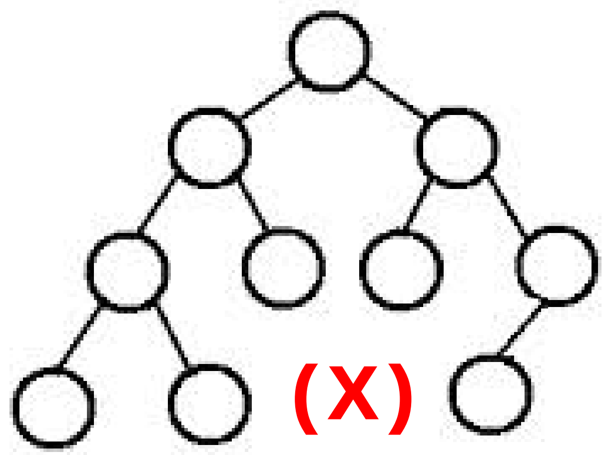
- Special binary trees

  (2) Complete binary tree

  A binary tree with n nodes and depth k is complete, if and only if its nodes corresponds to the nodes numbered from 1 to n in the full binary tree of depth k.



(O)



(X)

7.4 Basic operations

- Basic operations on tree

```
(1)BinTree Create (   )

(2)Boolean IsEmpty ( bt )

(3)BinTree MakeBT ( item, bt1, bt2 )

(4)element Data ( bt )

(5)BinTree Rchild ( bt )

(6)BinTree Lchild ( bt )
```

7.4 Basic operations

(1) Create ( )

– Create an empty binary tree

```
nptr Create ( )
{
    nptr nnode = (nptr) malloc ( sizeof(struct node) );

    nnode->data = EMPTY;
    nnode->lchild = nnode->rchild = NULL;

    return nnode;
}
```

7.4 Basic operations

(2) IsEmpty (bt)

– If bt is empty, then return TRUE;

```
boolean IsEmpty ( nptr bt )
{
    return ( bt->data == EMPTY );
}
```

7.4 Basic operations

(3) BinTree MakeBT (item, bt1, bt2)

– Return a binary tree whose data is item, lchild is bt1 and rchild is bt2

```
nptr MakeBT ( element item, nptr bt1, nptr bt2 )
{
    nptr nnode = (nptr) malloc ( sizeof(struct node) );

    nnode->data = item;
    nnode->lchild = bt1;
    nnode->rchild = bt2;

    return nnode;
}
```

7.4 Basic operations

(4) element Data (bt)

– Return data, if bt is neither NULL nor EMPTY

```
element Data (nptr bt )
{
    if ( bt == NULL )
      return ERROR;

    if ( IsEmpty ( bt ) )
      return EMPTY;

    return bt->data;
}
```

7.4 Basic operations

(5) BinTree Rchild (bt)

– Return right child of bt, if bt is not NULL

```
nptr Rchild (nptr bt )
{
    if ( bt == NULL )
      return ERROR;

    return bt->rchild;
}
```

7.4 Basic operations

(6) BinTree Lchild (bt)

– Return left child of bt, if bt is not NULL

```
nptr Lchild (nptr bt )
{
    if ( bt == NULL )
      return ERROR;

    return bt->lchild;
}
```

7.4 Basic operations

- Search (traversal)
  - Given a key and a tree, determine whether there is a node in the tree whose value coincides with the key
  - **An operation to visit all the nodes of a tree**
  - Two basic searches for a general tree
    - Depth-first search (DFS)
    - Breadth-first search (BFS)

7.4 Basic operations

- Search (traversal) on a binary tree
  - There are three combinations of visiting orders for a node and its child nodes

(1) Inorder traversal

- Left child node → root node → right child node

(2) Preorder traversal

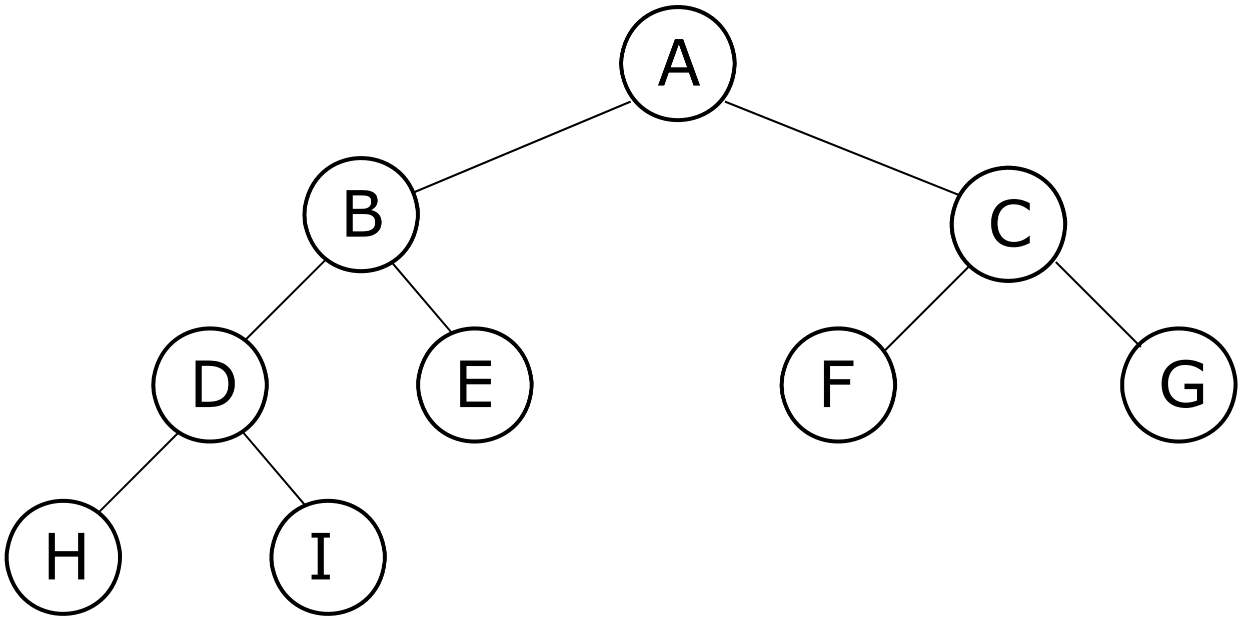
- Root node → left child node → right child node

(3) Postorder traversal

- Left child node → right child node → root node

7.4 Basic operations

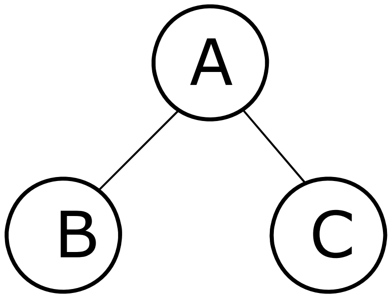
- Example binary tree



7.4 Basic operations

(1) Inorder: Left → Root → Right

```
void inorder ( nptr bt )
{
    if ( bt ) {
      inorder ( bt->lchild );
      print ( bt->data );
      inorder ( bt->rchild );
    }
}
```



B A C

7.4 Basic operations

Inorder *(A)*

Inorder *(B) A* ***Inorder*** *(C)*

Inorder *(D) B* ***Inorder*** *(E) A* ***Inorder*** *(C)*

Inorder *(H) D* ***Inorder*** *(I) B* ***Inorder*** *(E) A* ***Inorder*** *(C)*

H D ***Inorder*** *(I) B* ***Inorder*** *(E) A* ***Inorder*** *(C)*

H D I B ***Inorder*** *(E) A* ***Inorder*** *(C)*

H D I B E A ***Inorder*** *(C)*

H D I B E A ***Inorder*** *(F) C* ***Inorder*** *(G)*

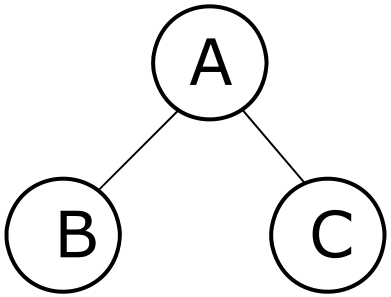
H D I B E A F C ***Inorder*** *(G)*

H D I B E A F C G

7.4 Basic operations

(2) Preorder: Root → Left → Right

```
void preorder ( nptr bt )
{
    if ( bt ) {
      print ( bt->data );
      preorder ( bt->lchild );
      preorder ( bt->rchild );
    }
}
```



A B C

7.4 Basic operations

***Preorder** (A)*

*A **Preorder** (B) **Preorder** (C)*

*A B **Preorder** (D) **Preorder** (E) **Preorder** (C)*

*A B D **Preorder** (H) **Preorder** (I) **Preorder** (E) **Preorder** (C)*

*A B D H **Preorder** (I) **Preorder** (E) **Preorder** (C)*

*A B D H I **Preorder** (E) **Preorder** (C)*

*A B D H I E **Preorder** (C)*

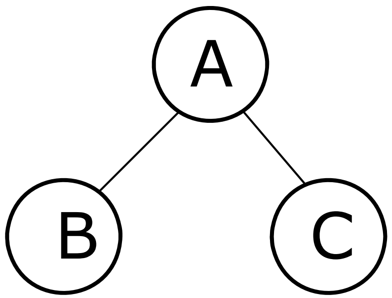
*A B D H I E C **Preorder** (F) **Preorder** (G)*

*A B D H I E C F **Preorder** (G)*

A B D H I E C F G

(3) Postorder: Left → Right → Root

```
void postorder ( nptr bt )
{
    if ( bt ) {
      postorder ( bt->lchild );
      postorder ( bt->rchild );
      print ( bt->data );
    }
}
```



B C A

7.4 Basic operations

***Postorder** (A)*

***Postorder** (B) **Postorder** (C) A*

***Postorder** (D) **Postorder** (E) B **Postorder** (C) A*

***Postorder** (H) **Postorder** (I) D **Postorder** (E) B **Postorder** (C) A*

*H **Postorder** (I) D **Postorder** (E) B **Postorder** (C) A*

*H I D **Postorder** (E) B **Postorder** (C) A*

*H I D E B **Postorder** (C) A*

*H I D E B **Postorder** (F) **Postorder** (G) C A*

*H I D E B F **Postorder** (G) C A*

H I D E B F G C A

7.5 Binary search tree

7.5.1 Definition

7.5.2 Searching a binary search tree

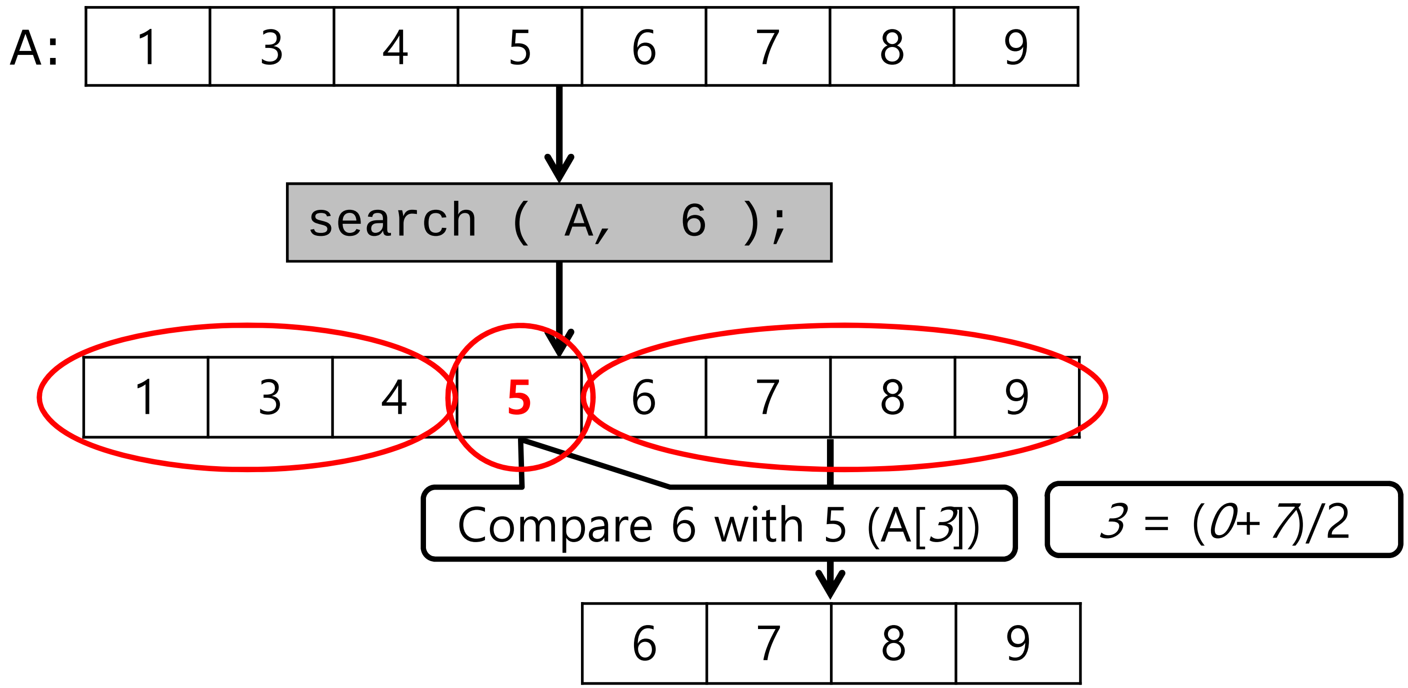
7.5.3 Inserting into a binary search tree

7.5.4 Deletion from a binary search tree

7.5.5 Time complexity on a binary search tree

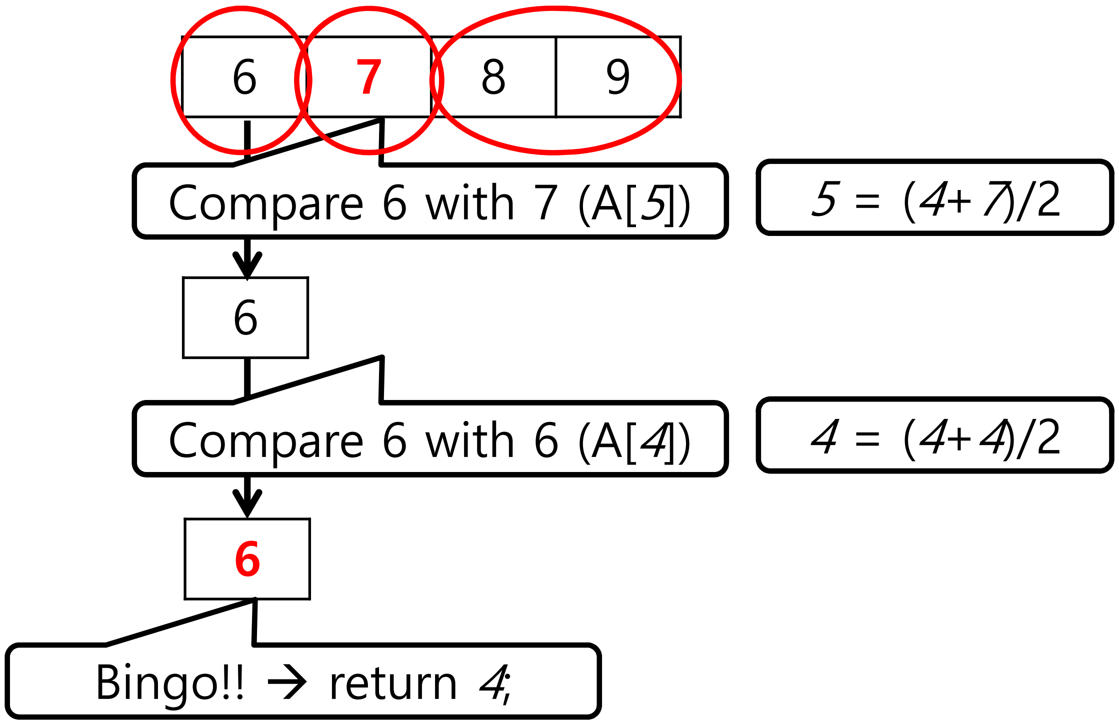
7.5.1 Definition

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



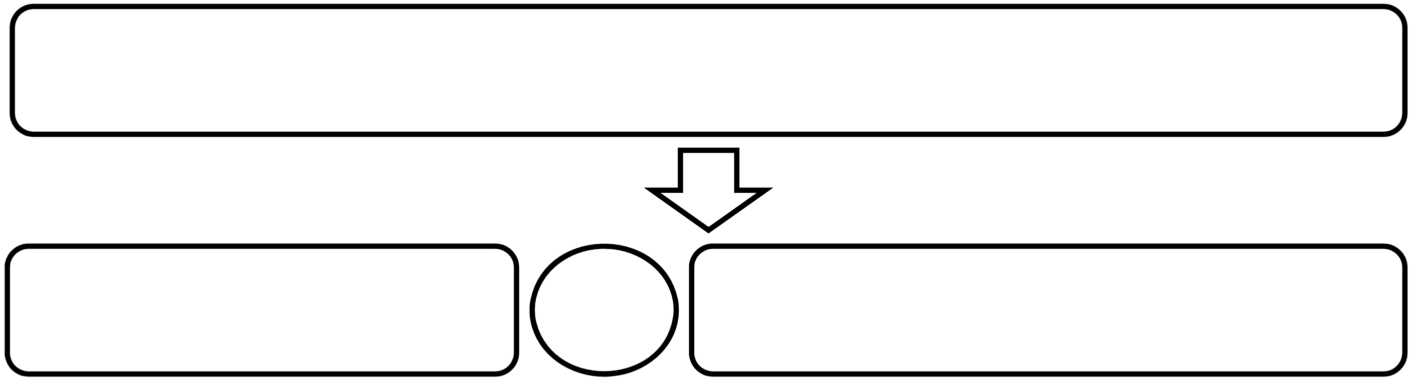
7.5.1 Definition

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



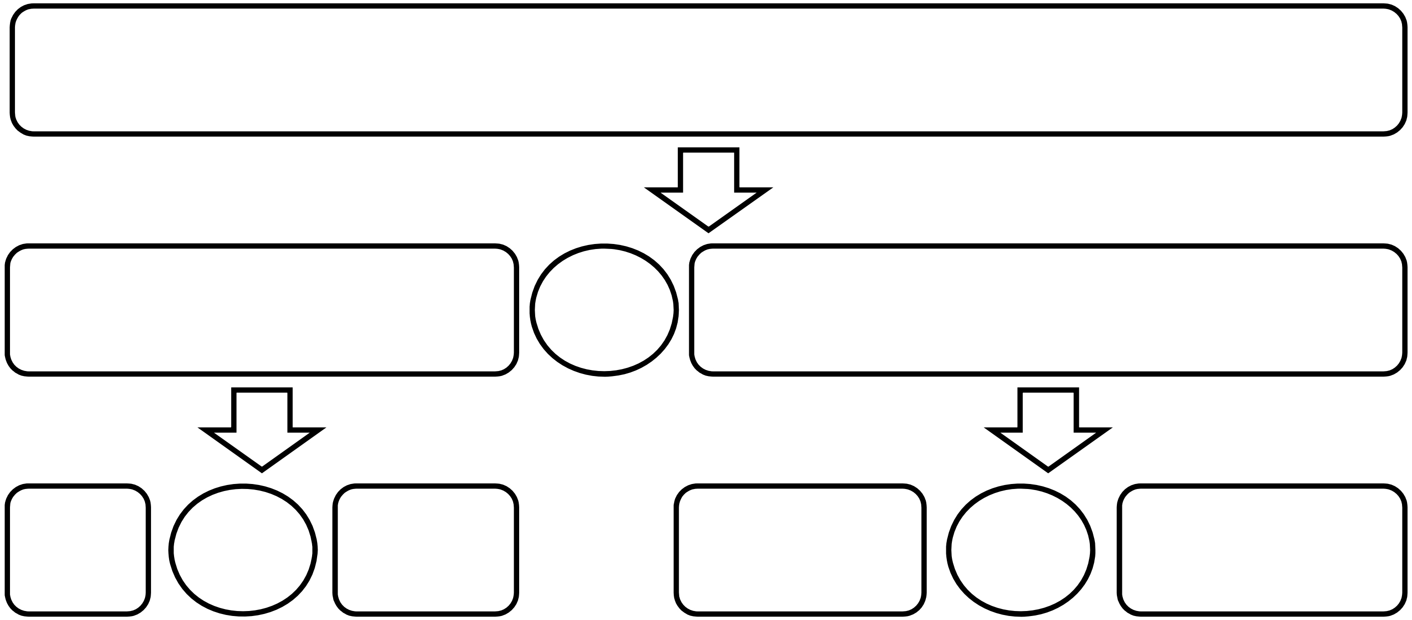
7.5.1 Definition

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



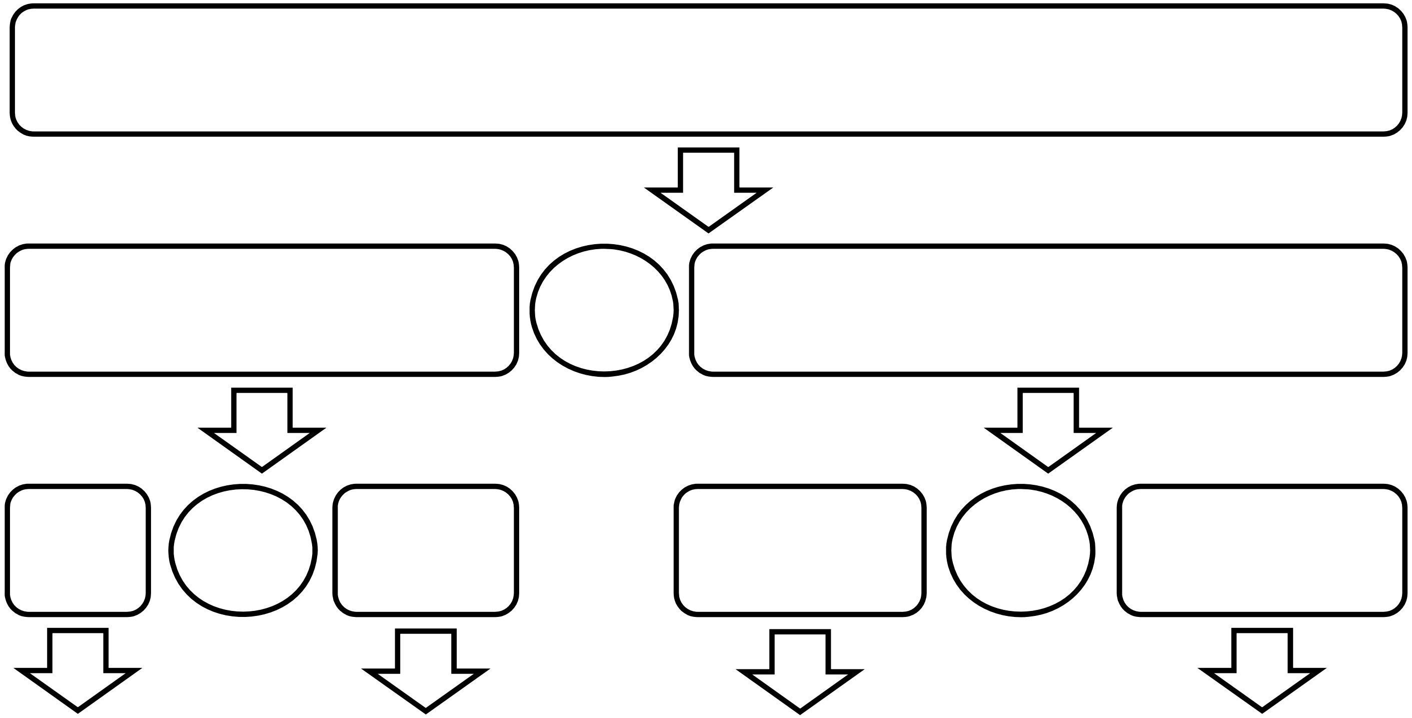
7.5.1 Definition

- 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**

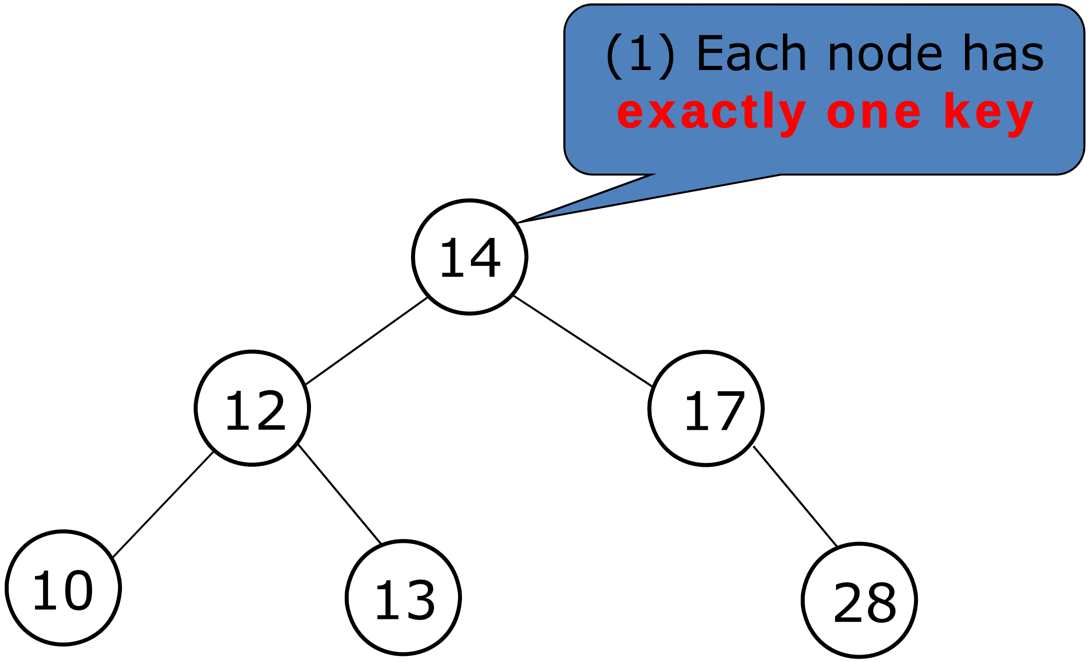
7.5.1 Definition

- 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						

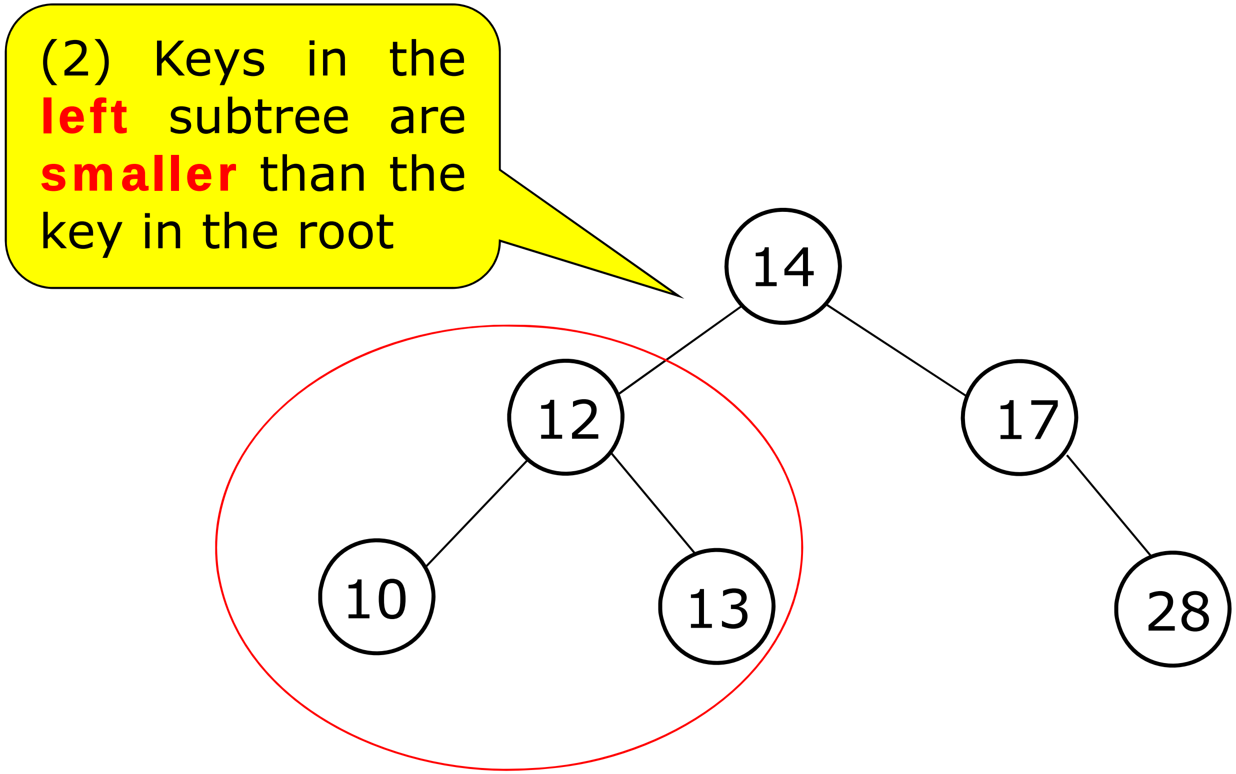
7.5.1 Definition

- Binary search tree



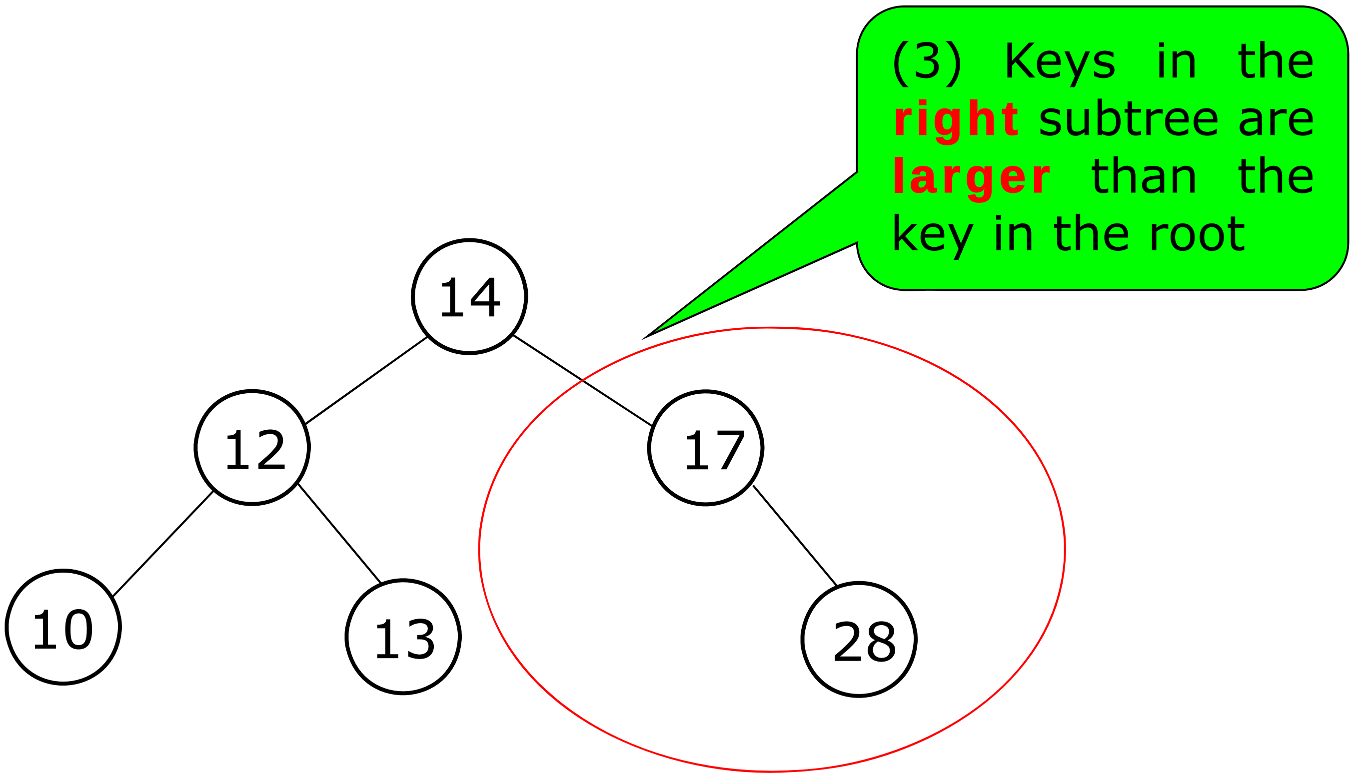
7.5.1 Definition

- Binary search tree



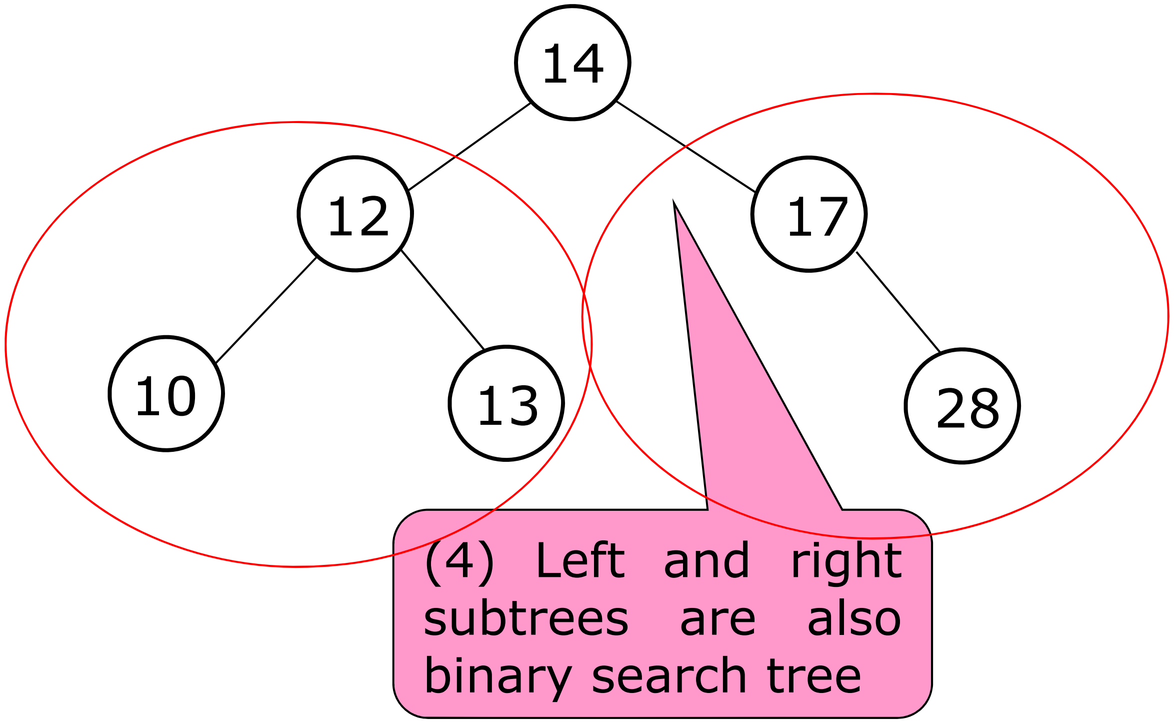
7.5.1 Definition

- Binary search tree



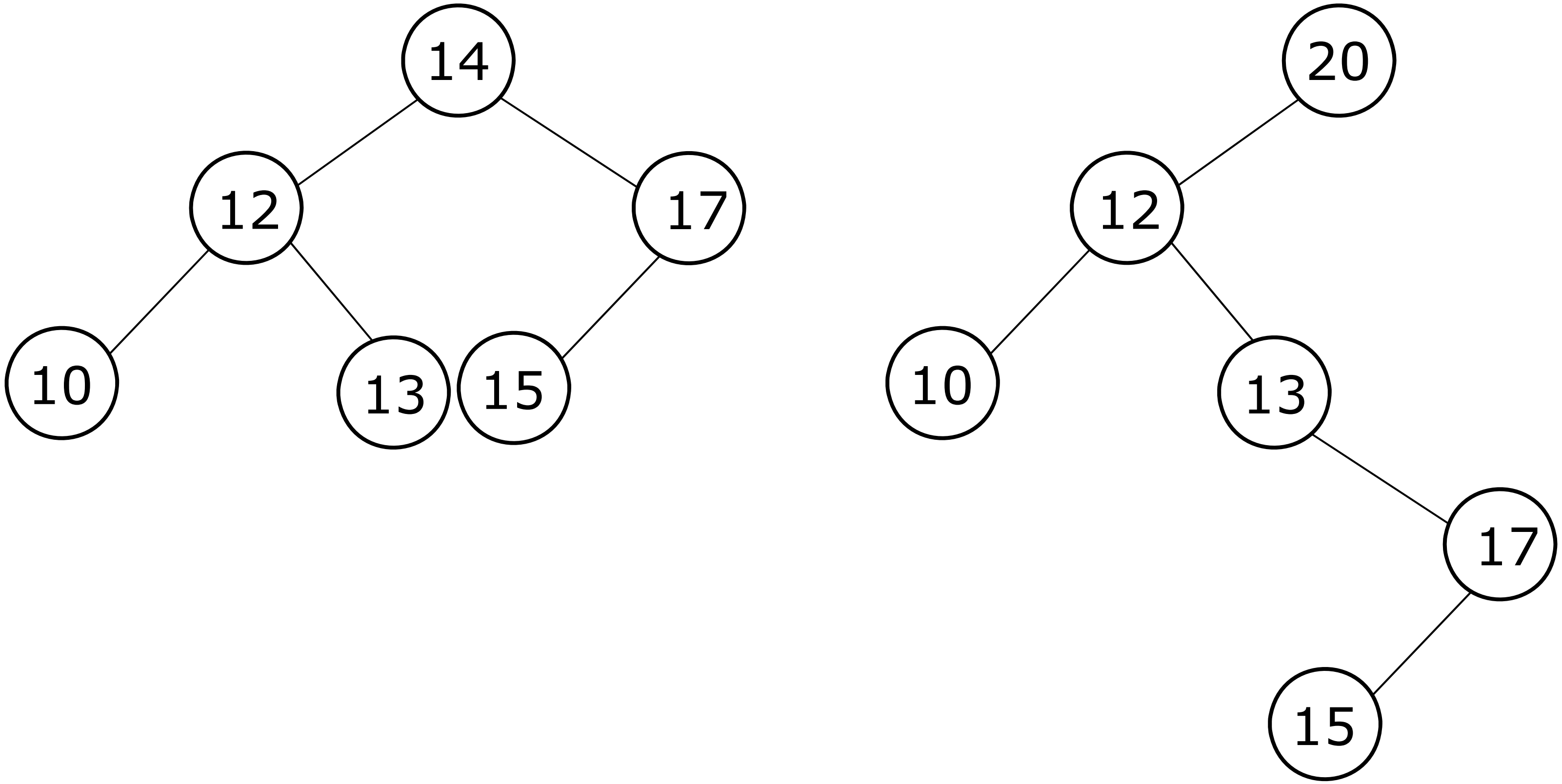
7.5.1 Definition

- Binary search tree



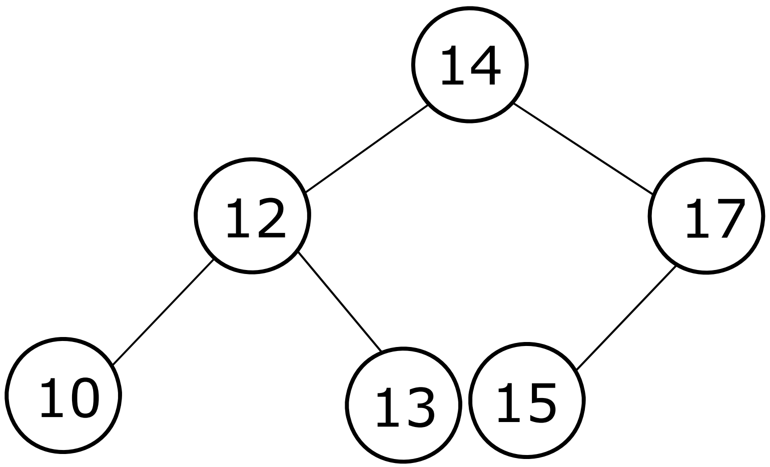
7.5.1 Definition

- Binary search trees



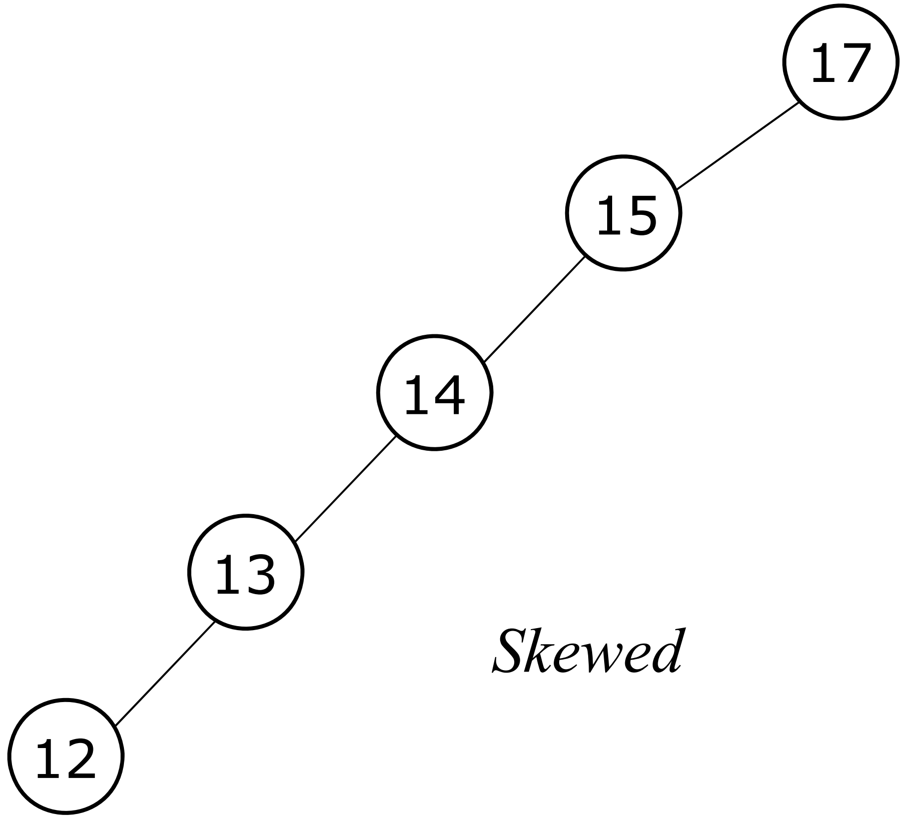
7.5.1 Definition

- Binary search trees (good and bad)



Balanced

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



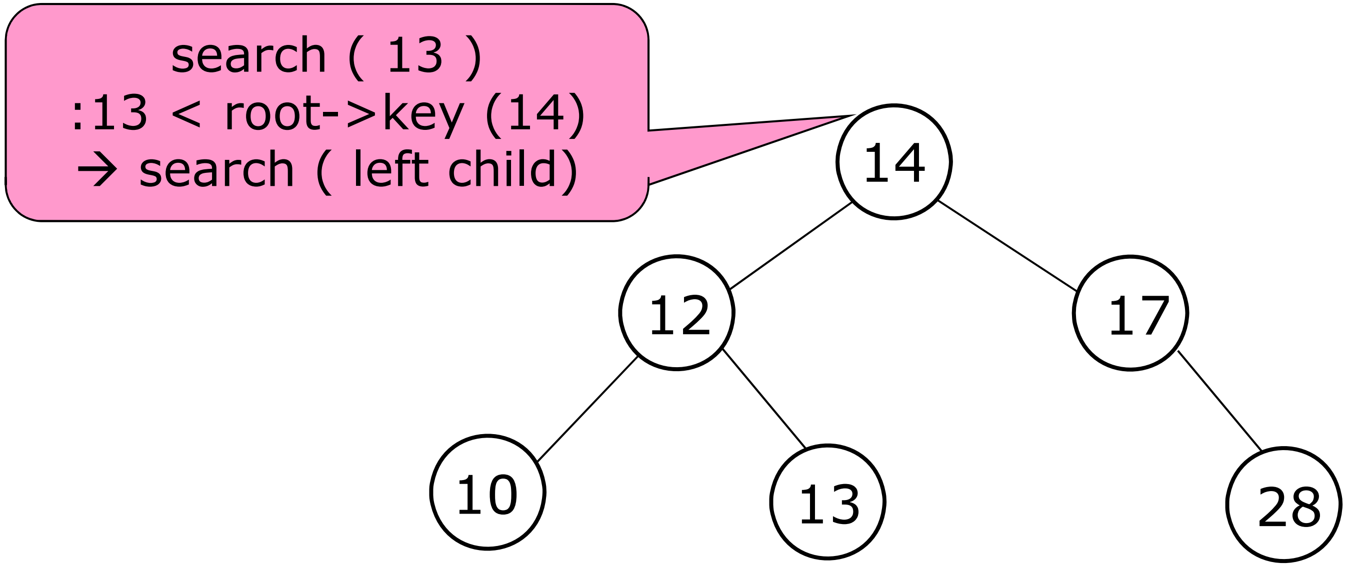
Skewed

7.5.2 Search

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

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

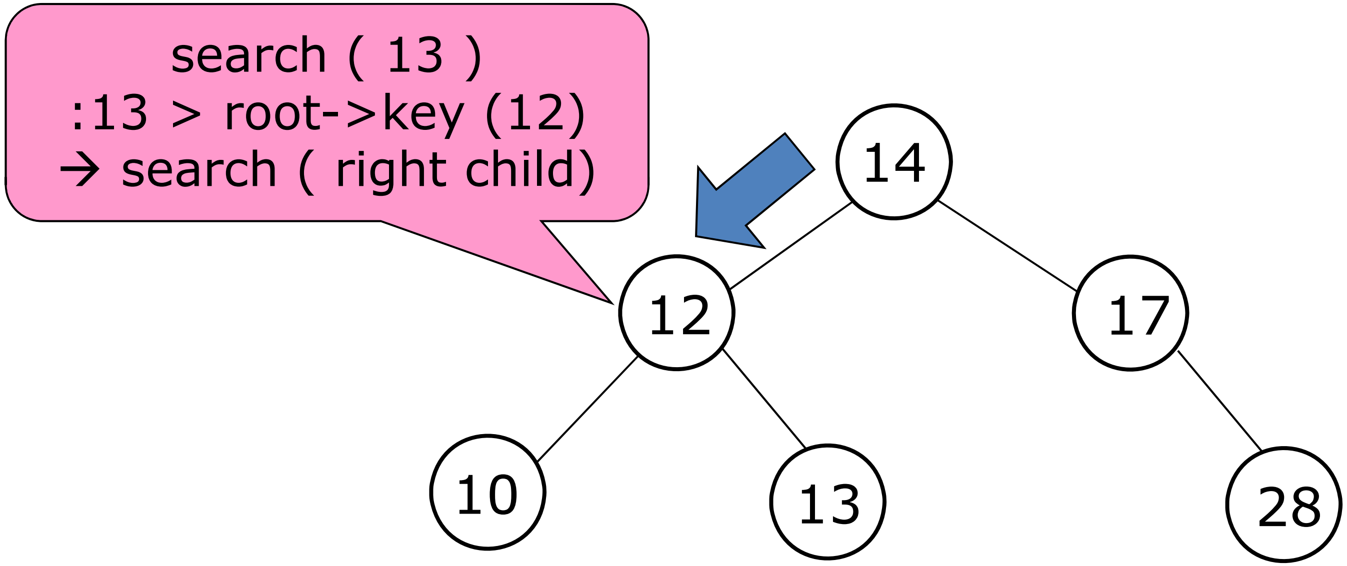
```
root->search ( 15 );
```



7.5.2 Search

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

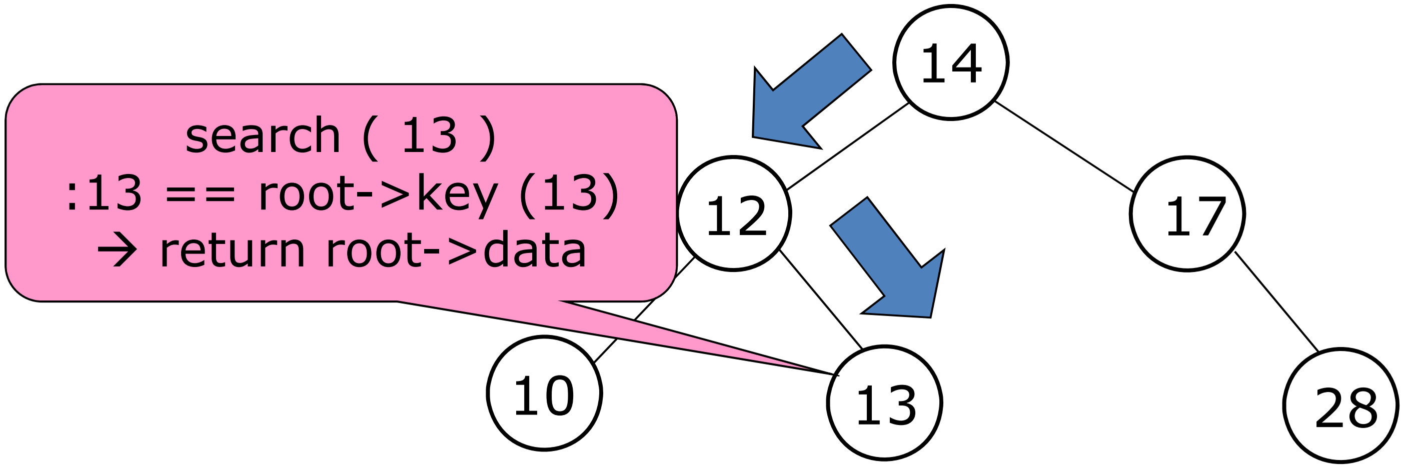
```
element node::search (KEY key )
```



7.5.2 Search

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

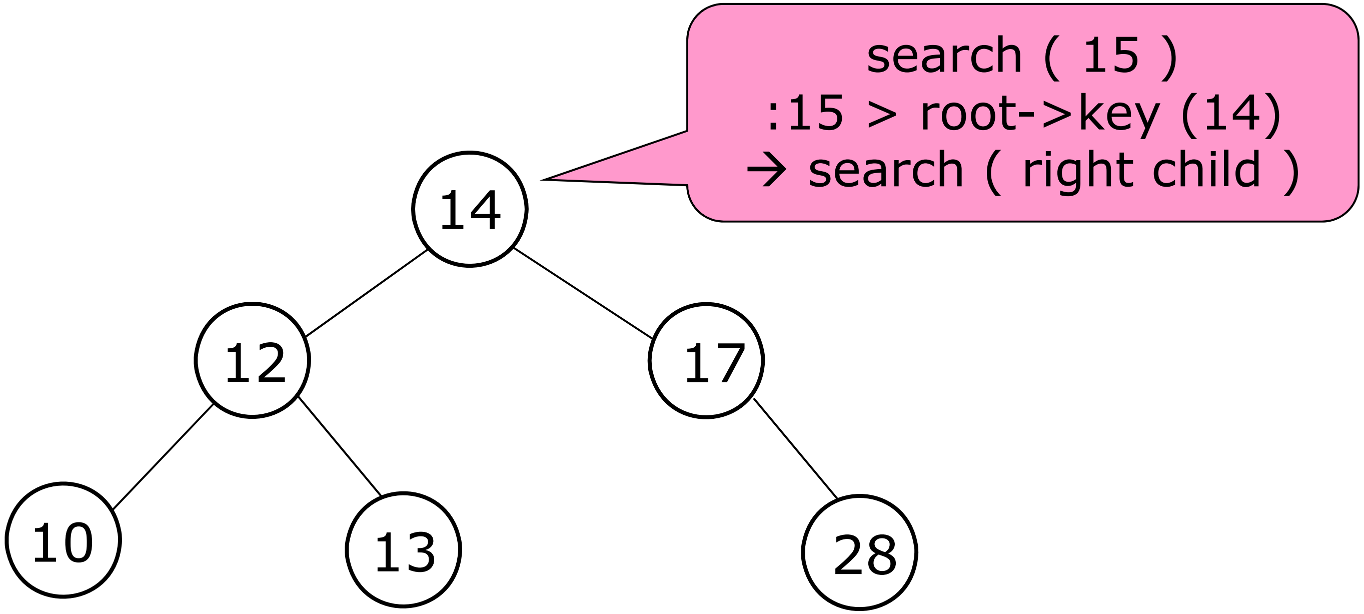
```
element node::search (KEY key )
```



7.5.2 Search

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

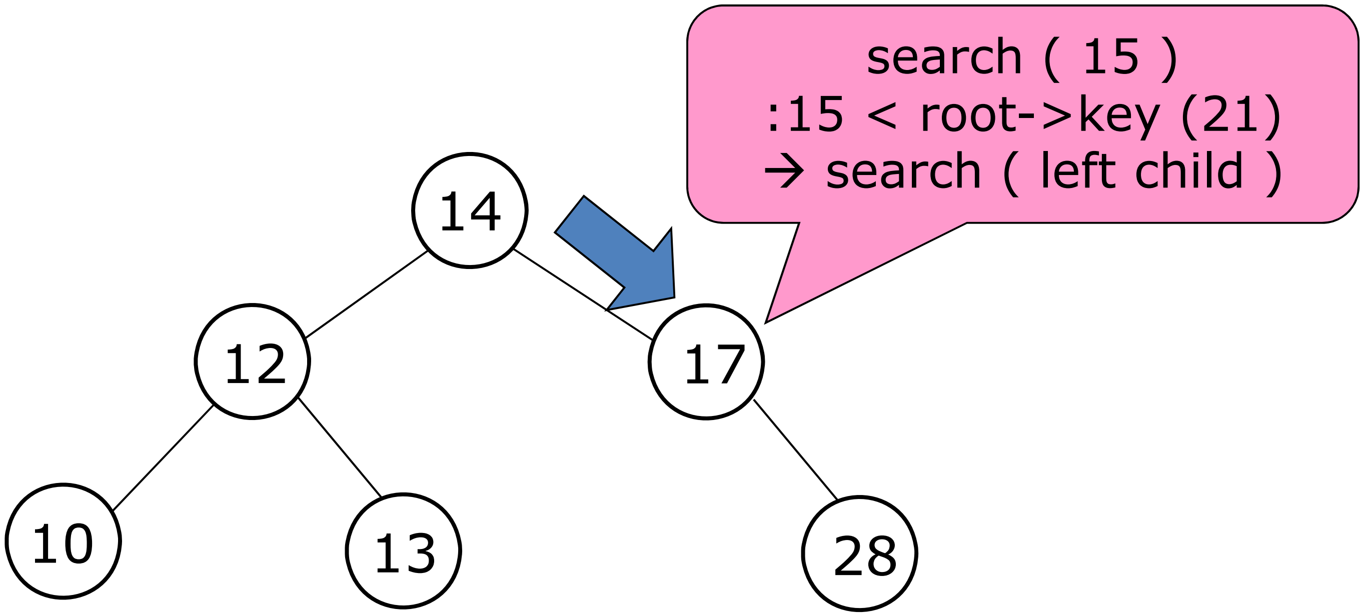
```
element node::search (KEY key )
```



7.5.2 Search

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

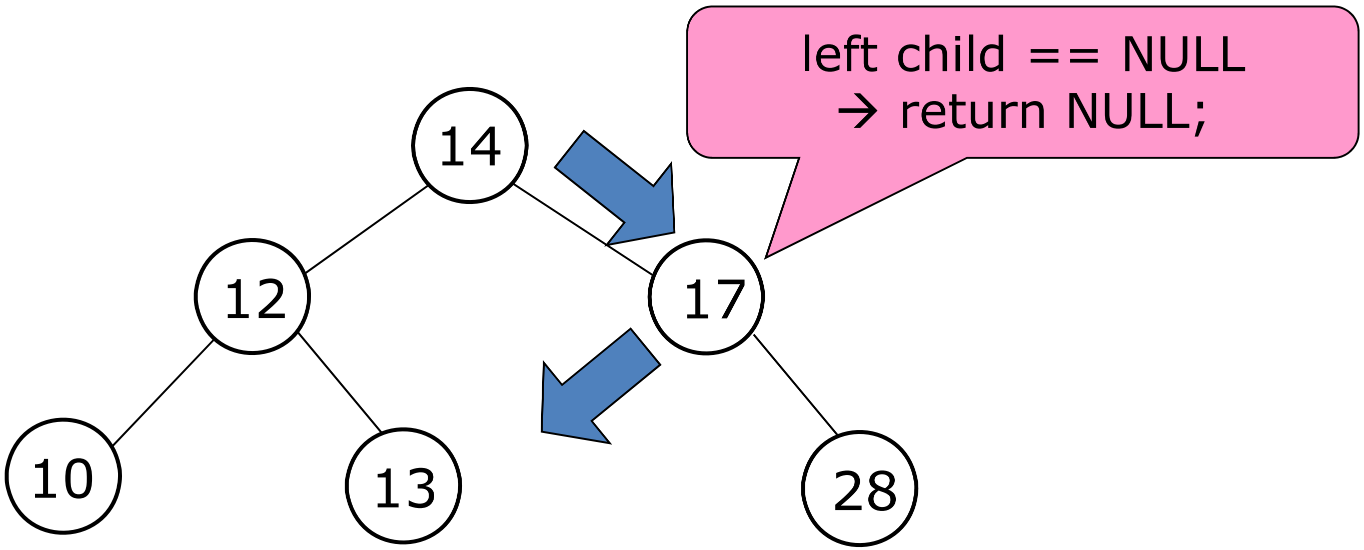
```
element node::search (KEY key )
```



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

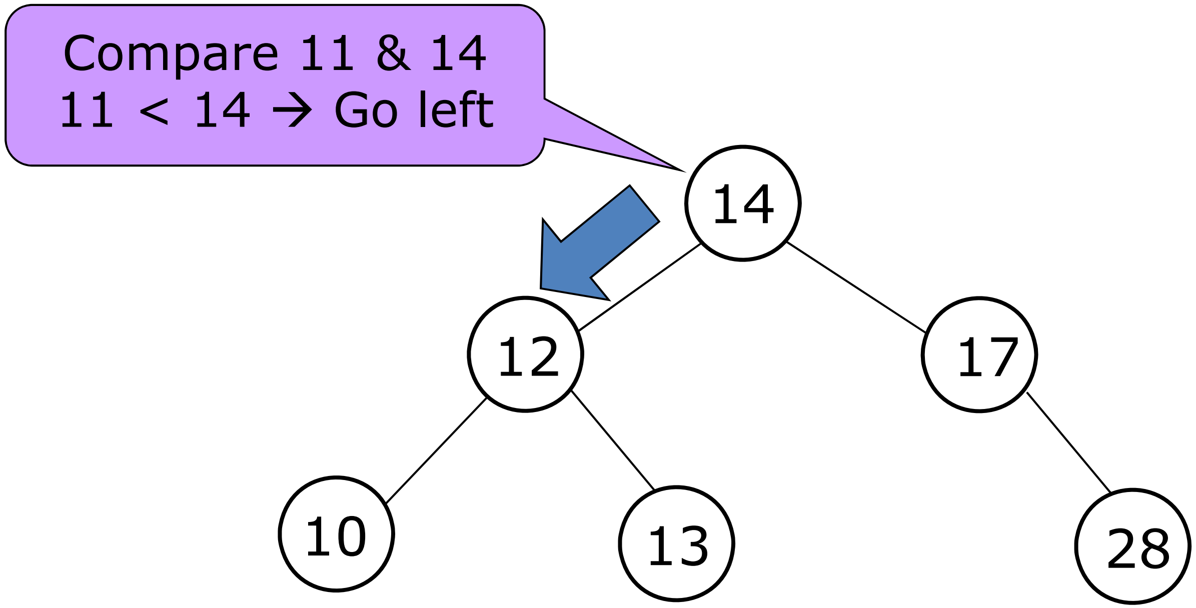
```
element node::search ( KEY key )
{
//  1. if this node has the key, then return this node’s data
    if ( key == this->key )
        return this->data;
//  2. if key < this->key, then search left subtree
//      else search right subtree
    if ( key < this->key )
        return search ( this->lchild, key );
    else
        return search ( this->rchild, key );
}
```

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

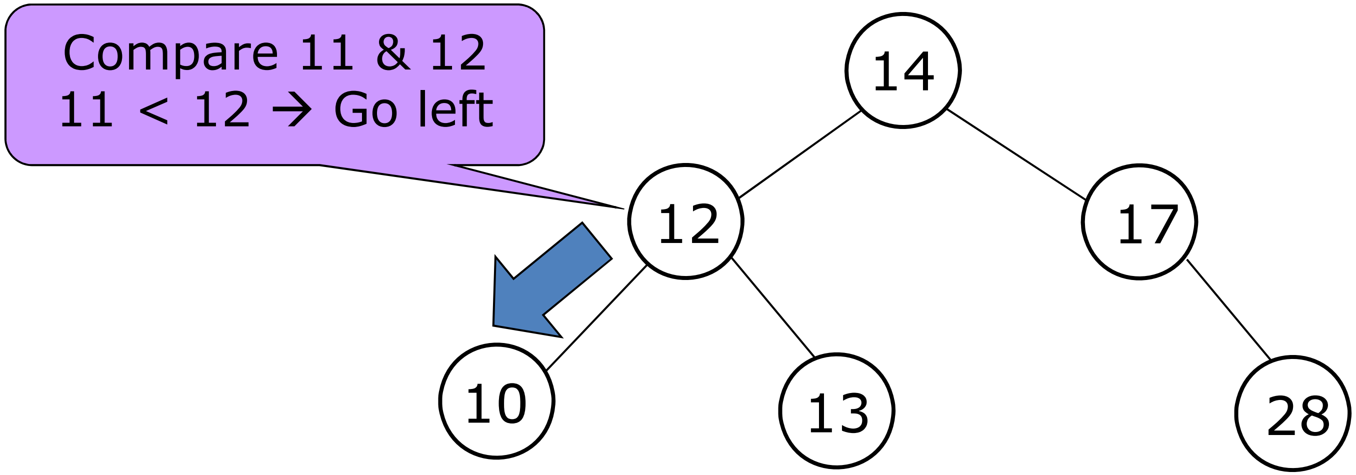
7.5.3 Insert

- Example
  - Insert <11>



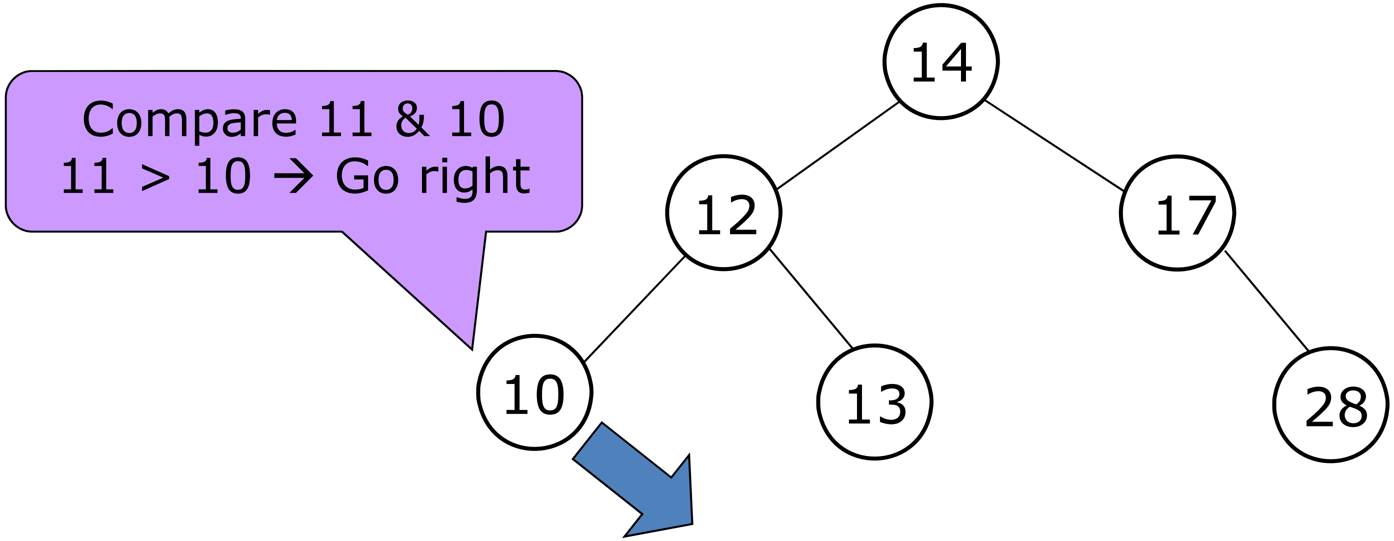
7.5.3 Insert

- Example
  - Insert <11>



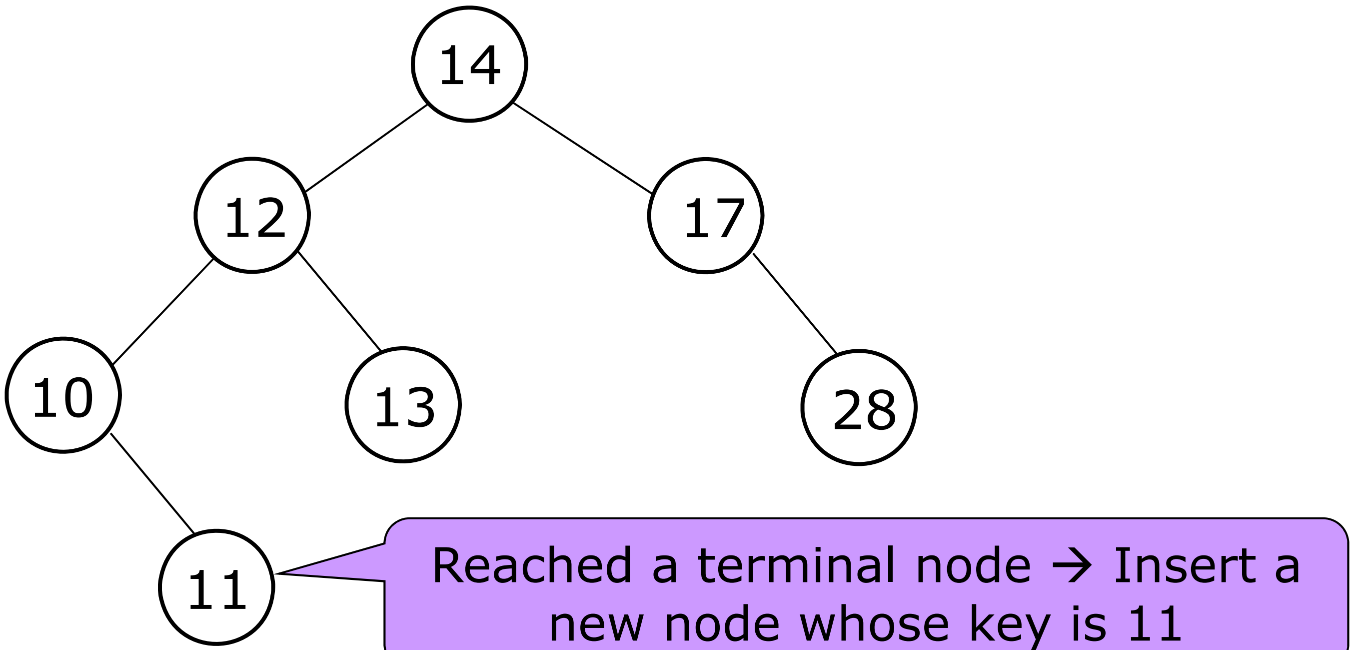
7.5.3 Insert

- Example
  - Insert <11>



7.5.3 Insert

- Example
  - Insert <11>



7.5.4 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

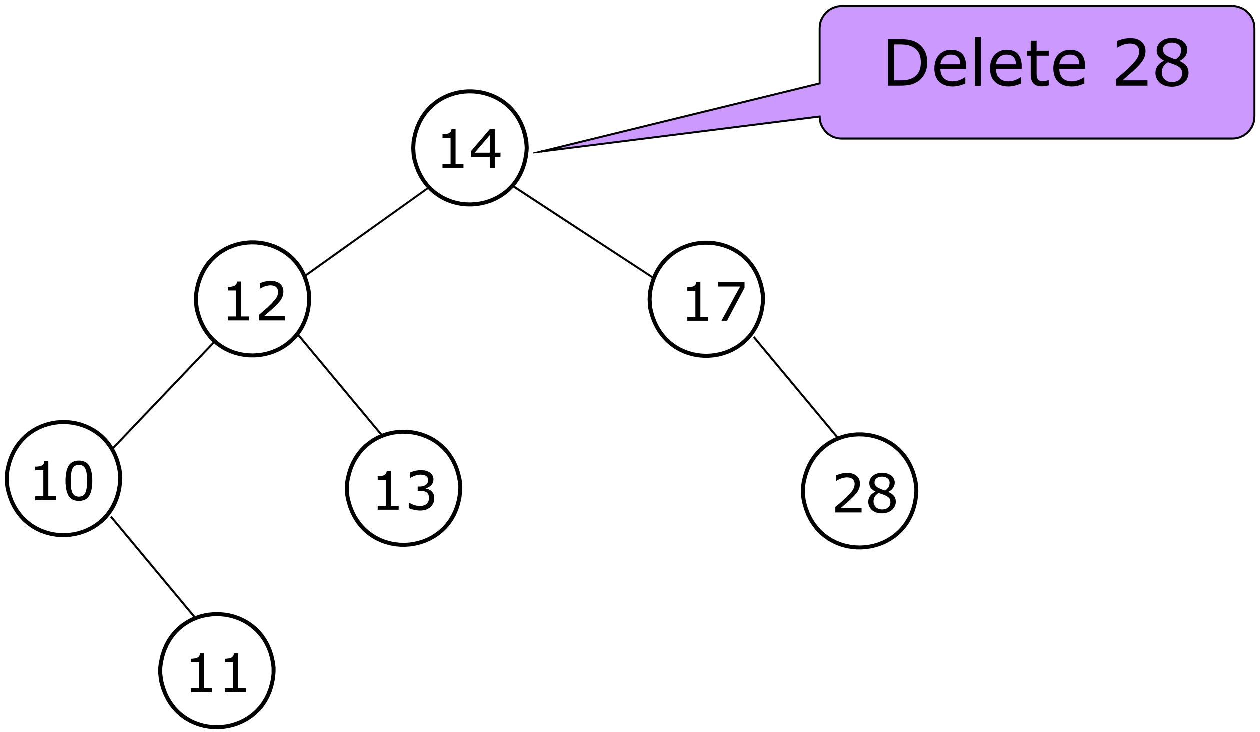
7.5.4 Delete

- Deleting leaf nodes
  → Delete the node

7.5.4 Delete

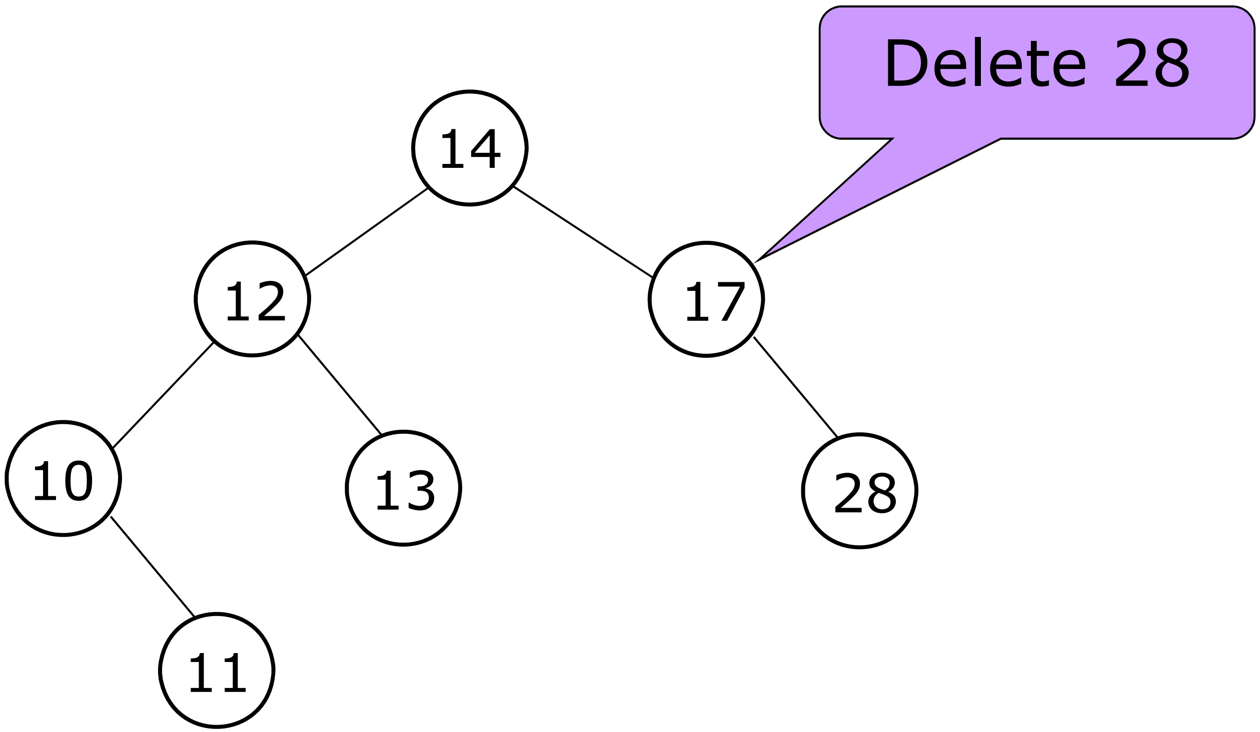
- Deleting leaf nodes

  → Delete the node



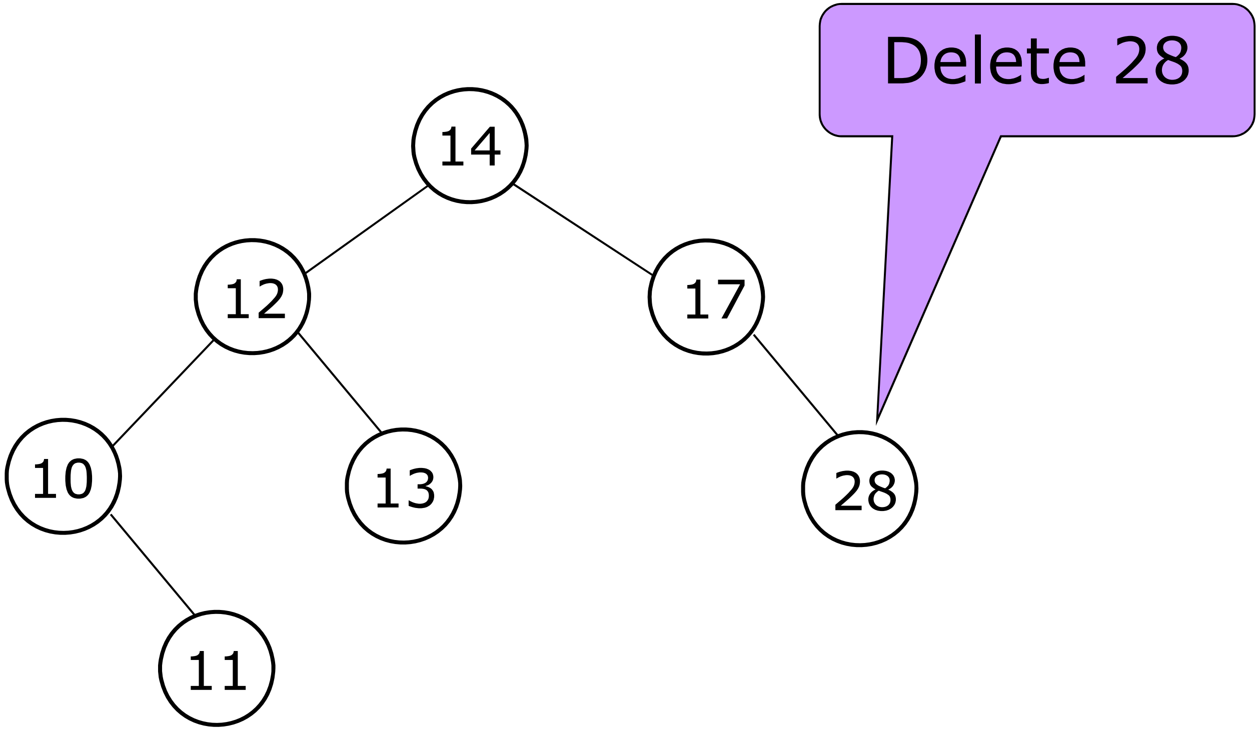
7.5.4 Delete

- Deleting leaf nodes
  → Delete the node



7.5.4 Delete

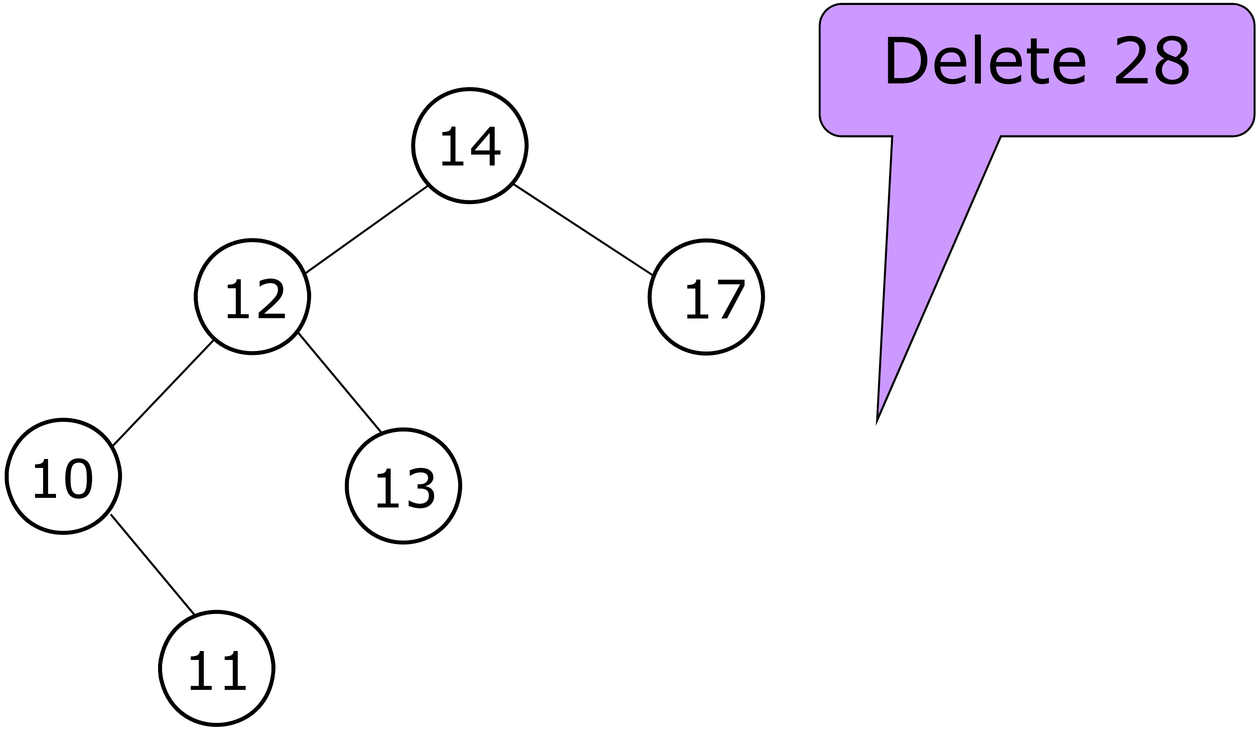
- Deleting leaf nodes
  → Delete the node



7.5.4 Delete

- Deleting leaf nodes

  → Delete the node



7.5.4 Delete

- Deleting internal nodes of one child

  → (1) Delete the node

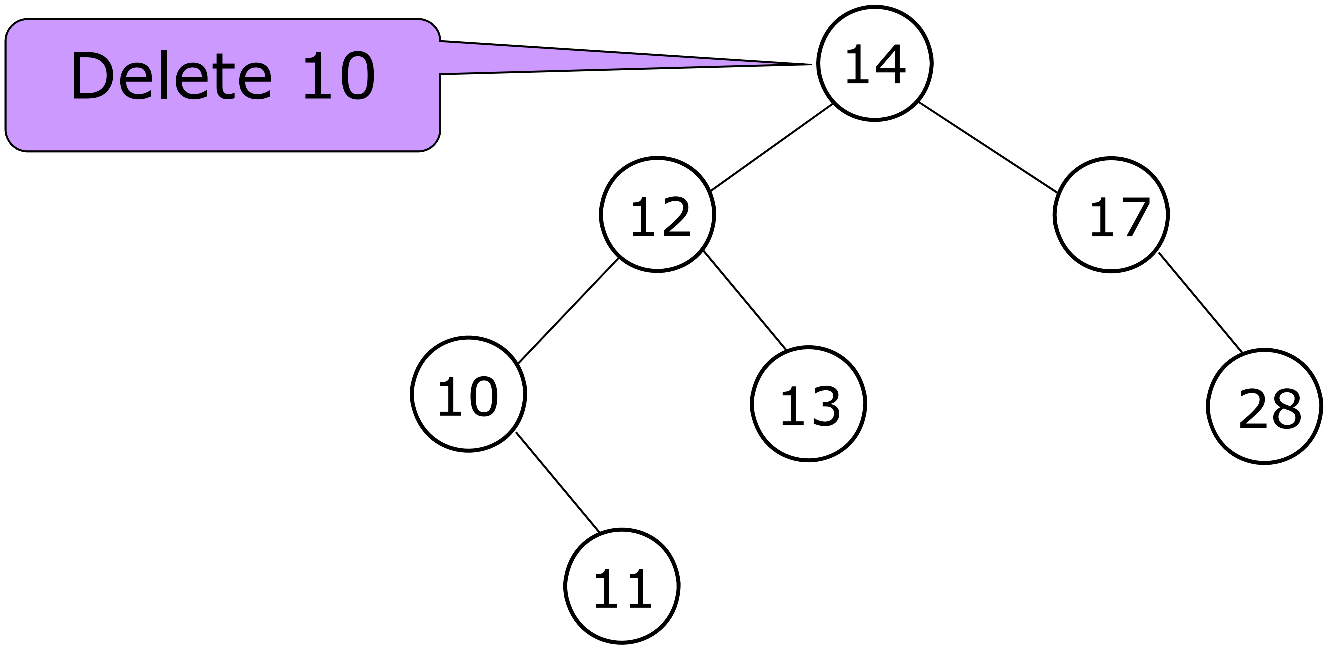
  (2) Make the child take place of the deleted node

7.5.4 Delete

- Deleting internal nodes of one child

  → (1) Delete the node

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

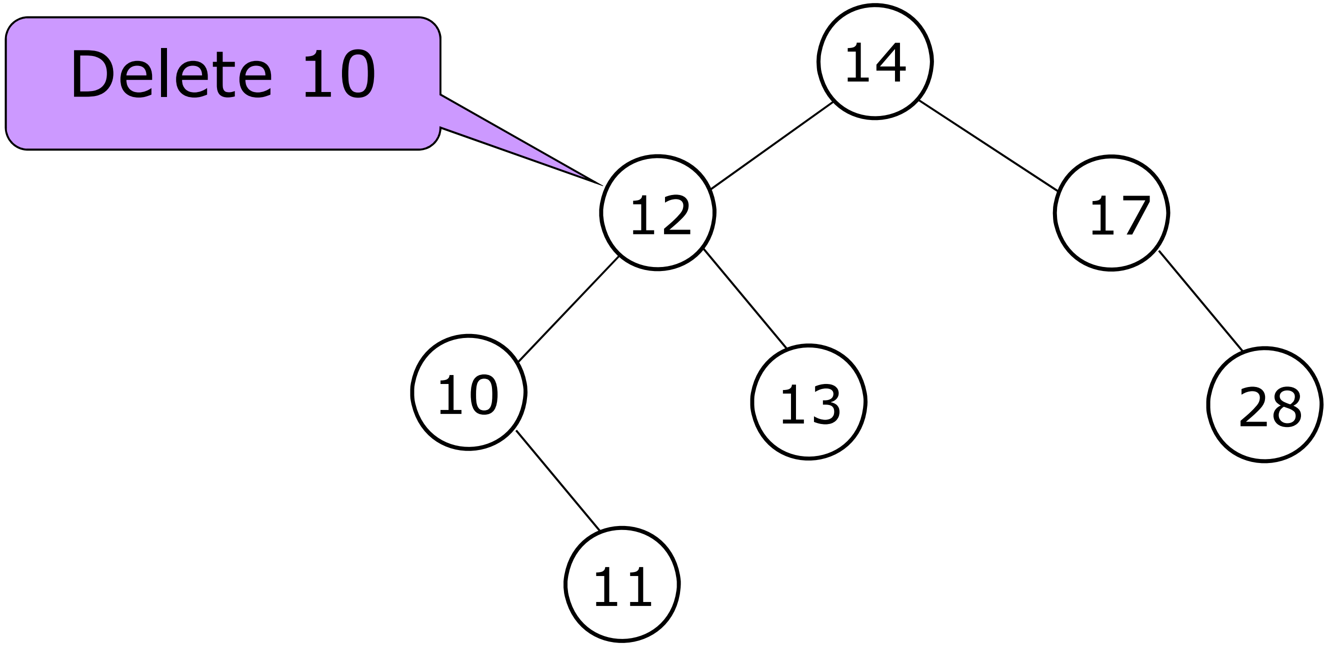


7.5.4 Delete

- Deleting internal nodes of one child

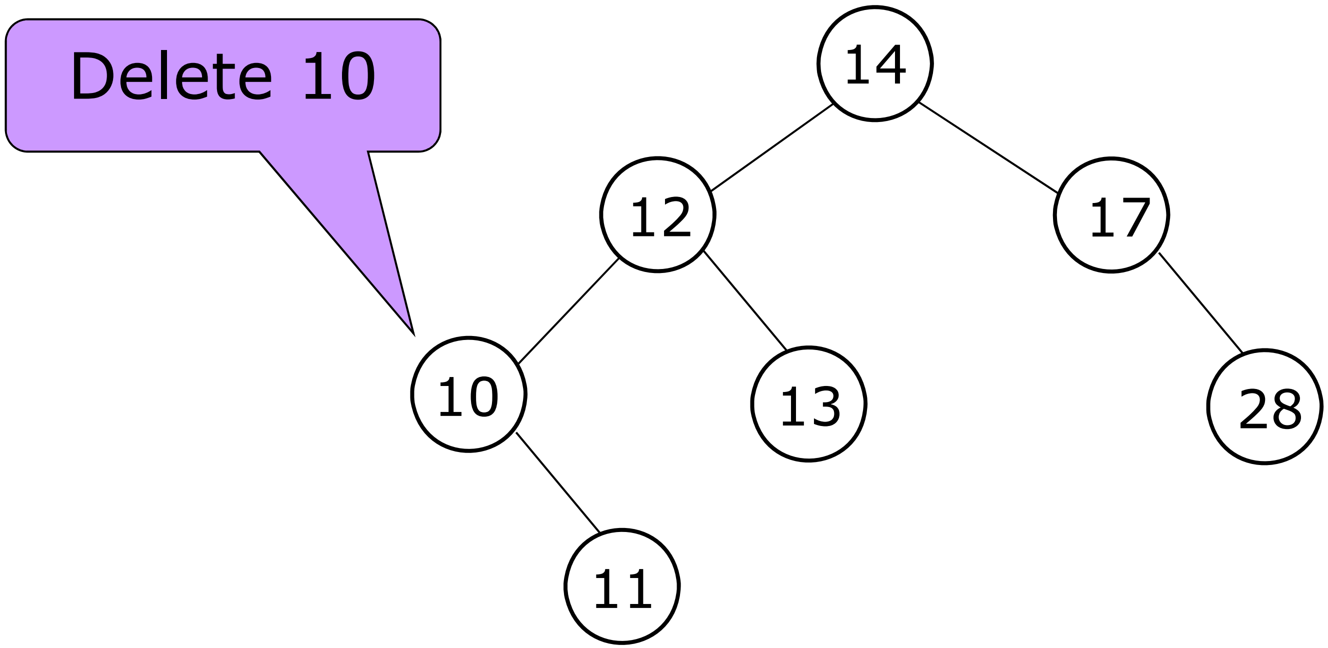
  → (1) Delete the node

  (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

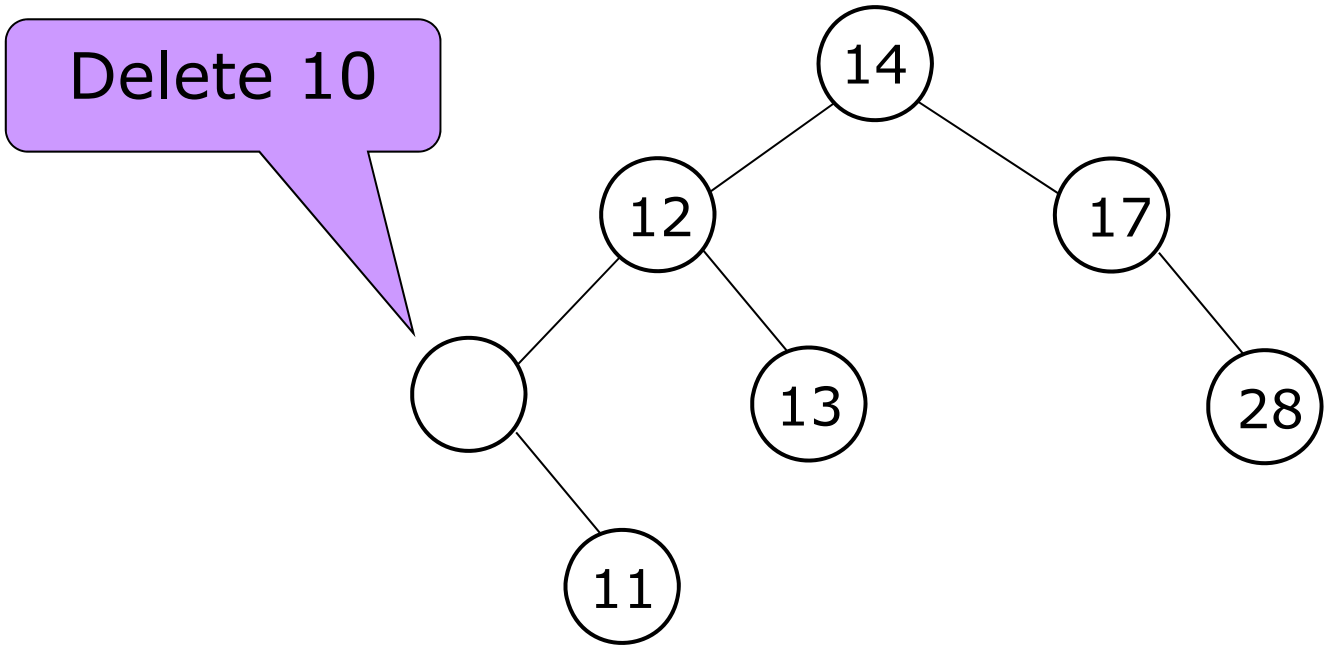


7.5.4 Delete

- Deleting internal nodes of one child

  → (1) Delete the node

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

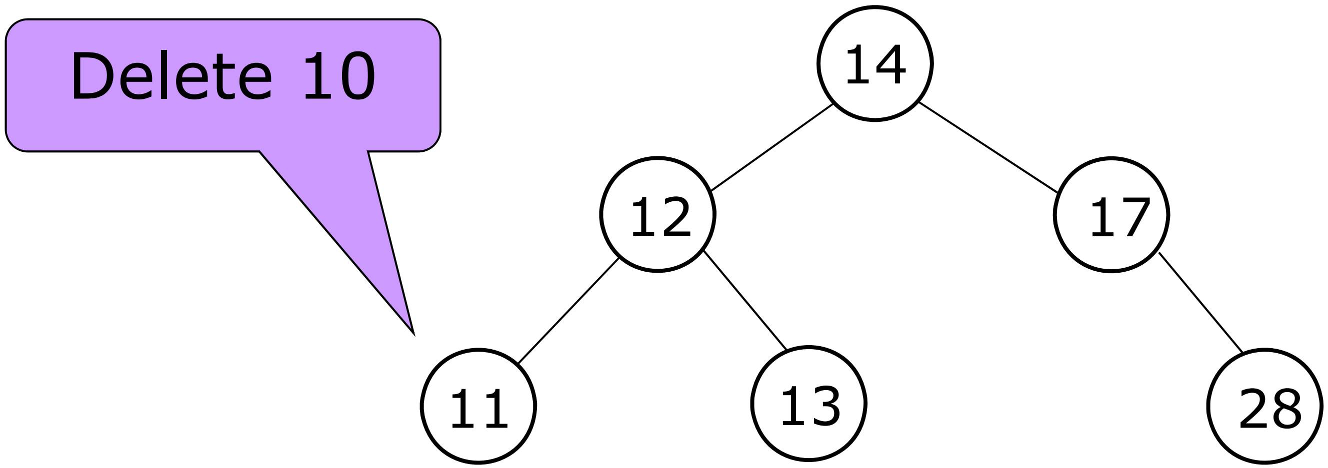


7.5.4 Delete

- 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

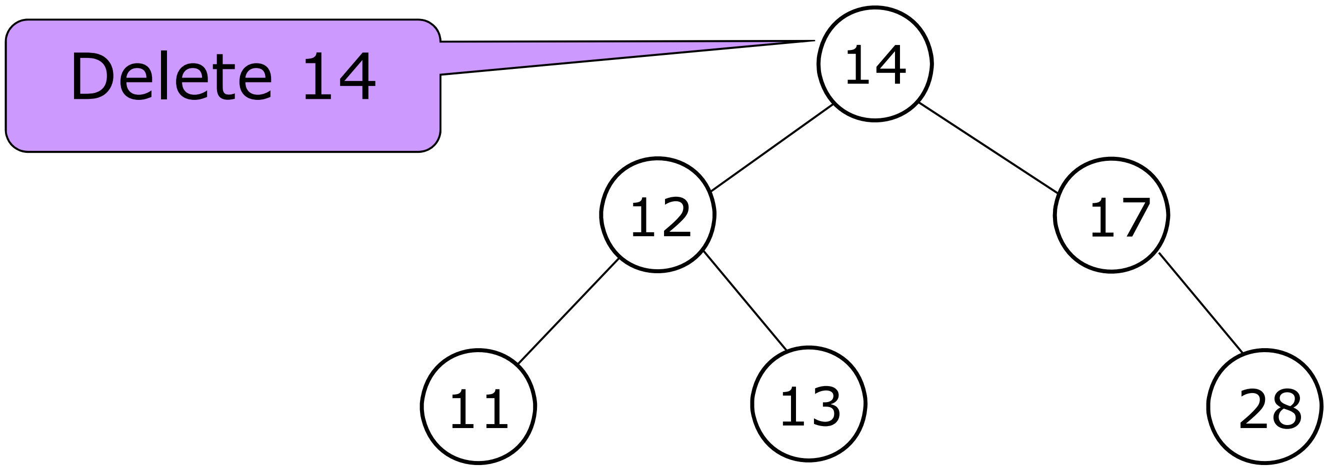
  (2) Move the maximum of its left subtree

  (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

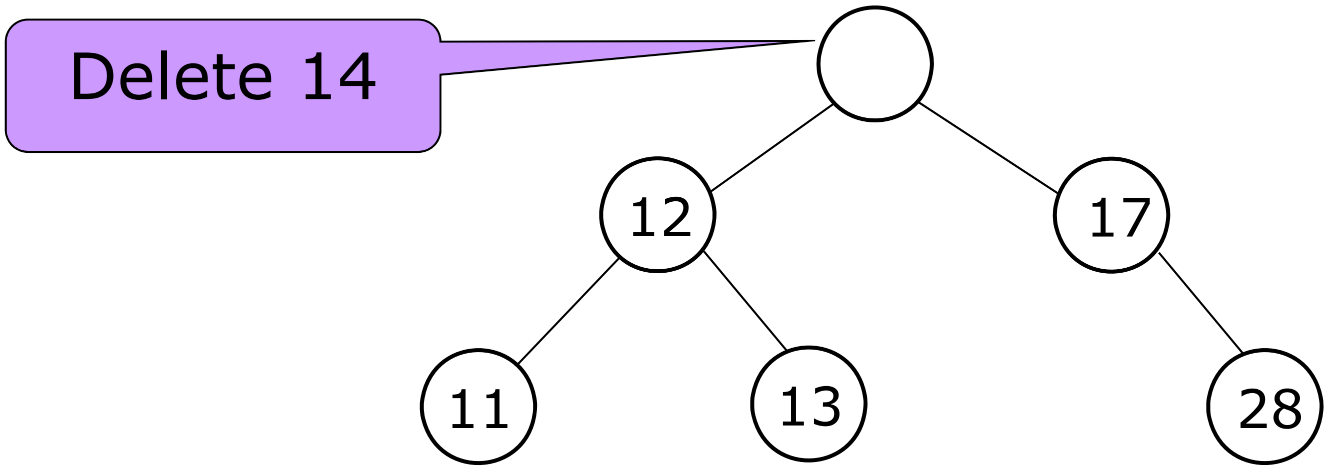


7.5.4 Delete

- 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

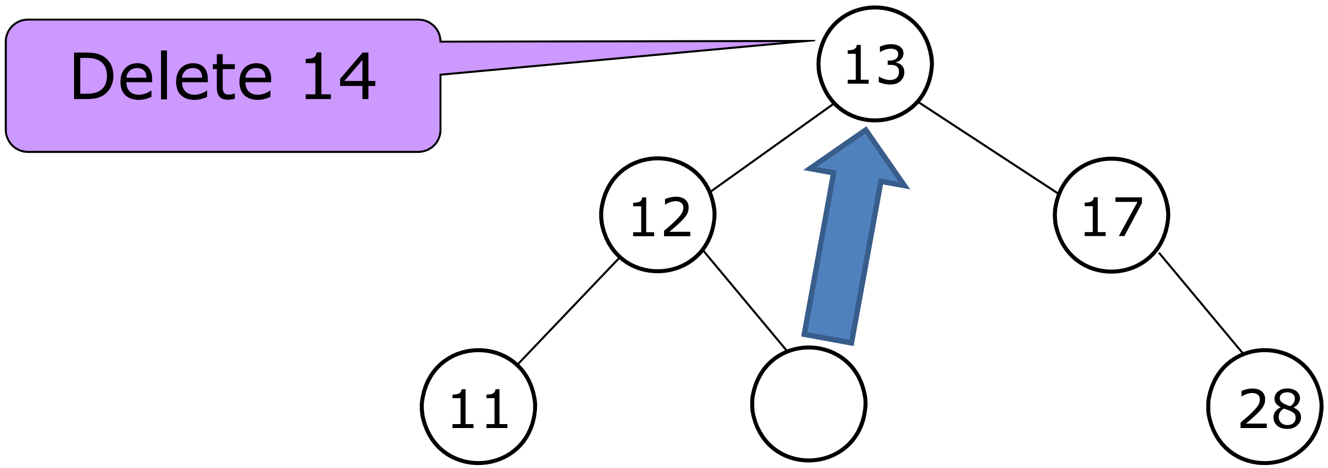


7.5.4 Delete

- 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

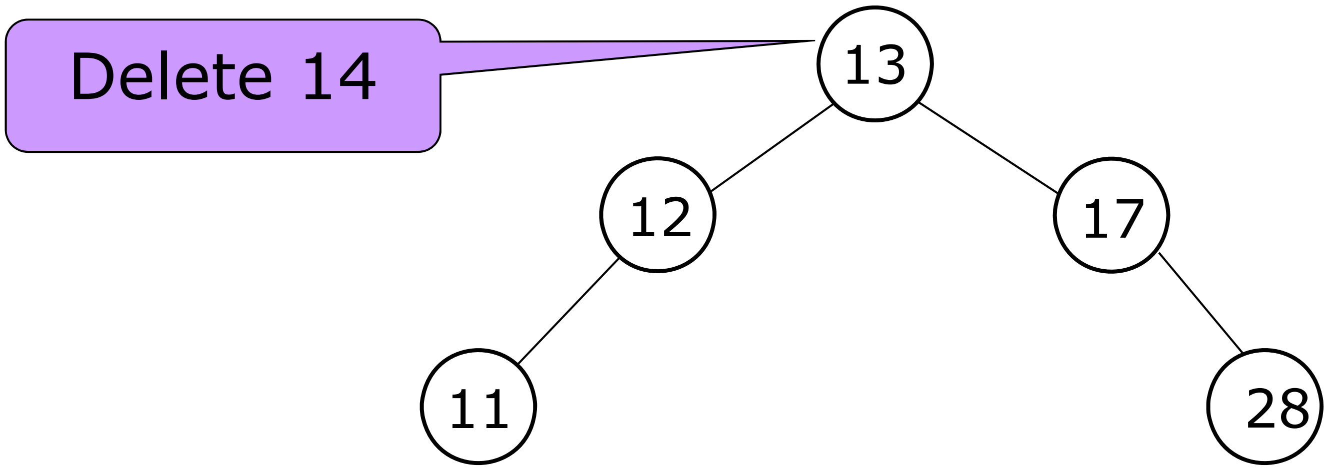


7.5.4 Delete

- Deleting internal nodes with two childs

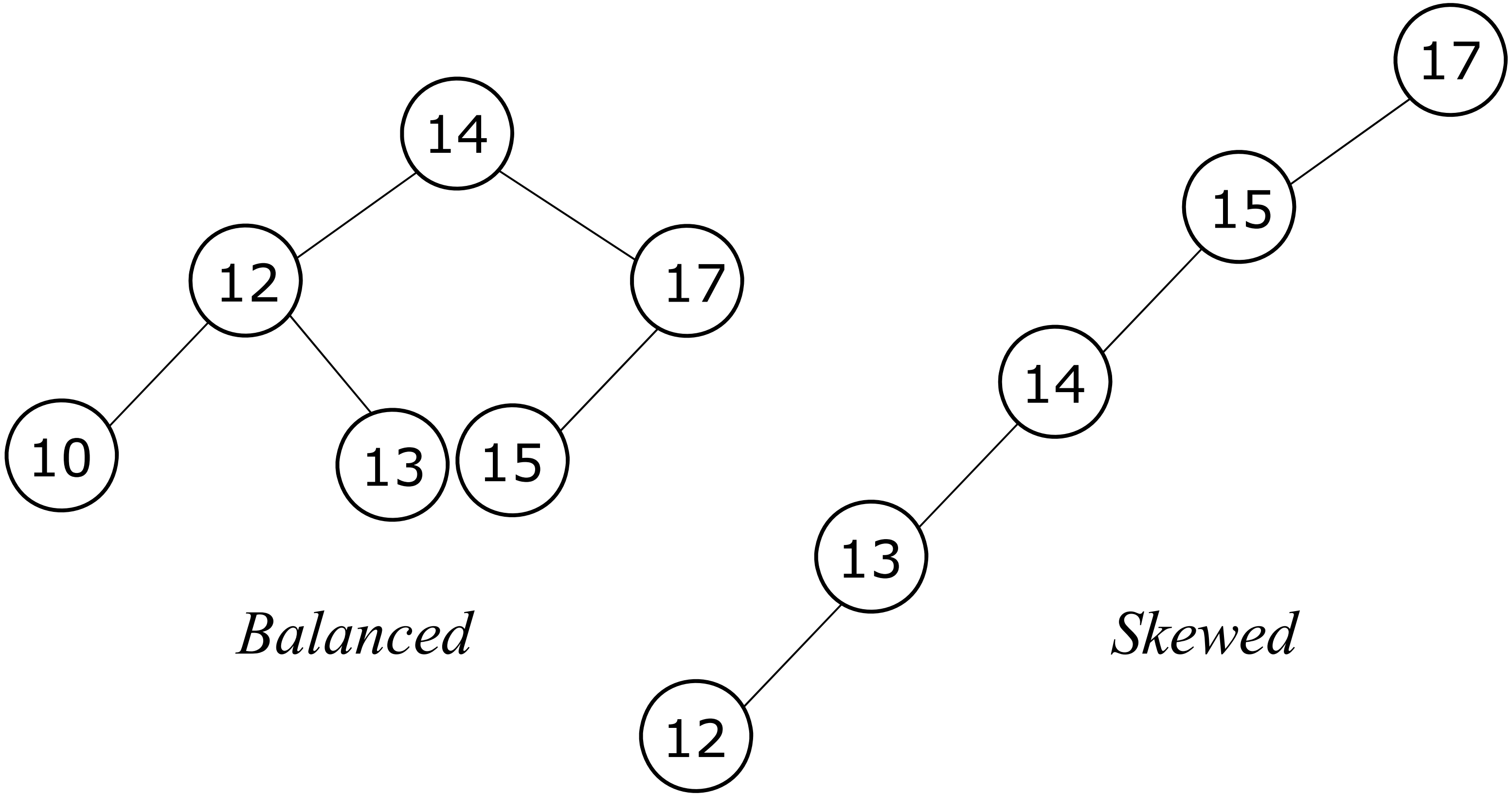
  → (1) Delete the node

  (2) Move the maximum of its left subtree (or the minimum of its right subtree) to the node



7.5.5 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
    - ……

7.6 Heap (히입)

7.6.1 Priority Queue

7.6.2 Definition of a Heap

7.6.3 Insertion into a Heap

7.6.4 Deletion from a Heap

7.6.1 Priority Queue

- Priority queue
  - The element to be deleted is the one with the highest (or lowest) priority
  - Example) Emergency room in hospital

7.6.1 Priority Queue

- Operations of priority queue
  - Push
    - Add a new element to the queue
    - Determine the position according to its priority
  - Pop
    - Remove the element of highest priority from the queue
  - Top
    - Search the element of the highest priority from the queue (do not remove the element)

7.6.1 Priority Queue

- Implementation of a priority queue using a sorted list
  - Push
    - Insert an element to a sorted list
  - Pop
    - Remove the first element from the list
  - Top
    - Return the first element of the list
  - Ex) Insert 16, 10, 33, 4 to the queue

4	**10**	**16**	**33**						

7.6.1 Priority Queue

- Implementation of a priority queue using a sorted array
  - Push: O(n)
  - Pop: O(n)
  - Top: O(1)

- Can we improve this?

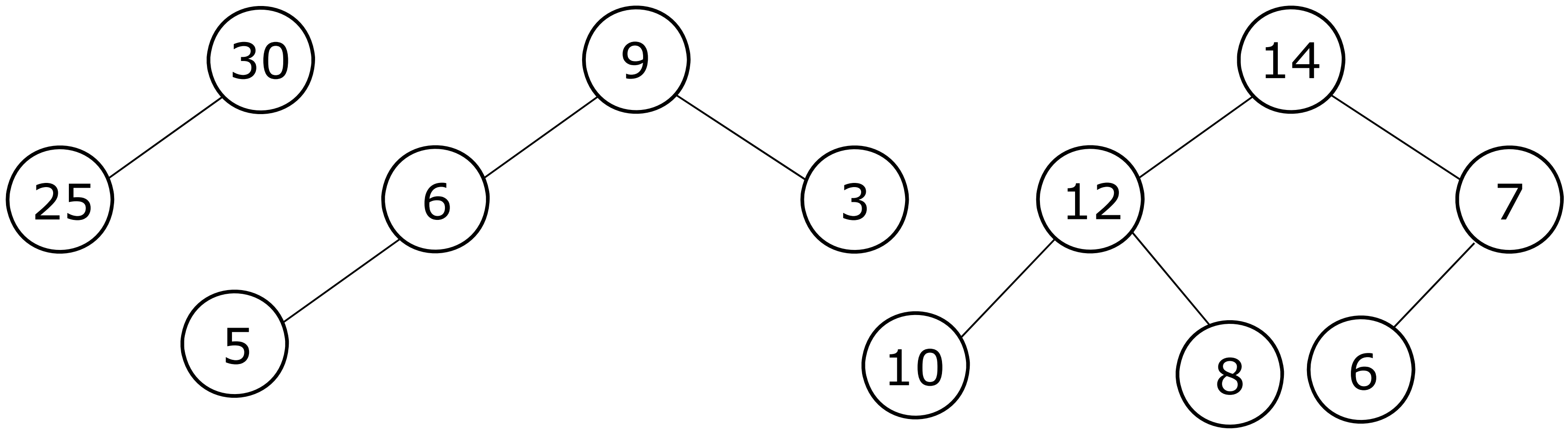
  → use tree!! (heap)

7.6.2 Definition of a Heap

- Heap
  - A tree-based implementation of a priority queue
  - A complete binary tree
  - Max heap
    - The key value in each node is no ***smaller*** than the key values of its child nodes
  - Min heap
    - The key value in each node is no ***larger*** than the key values of its child nodes

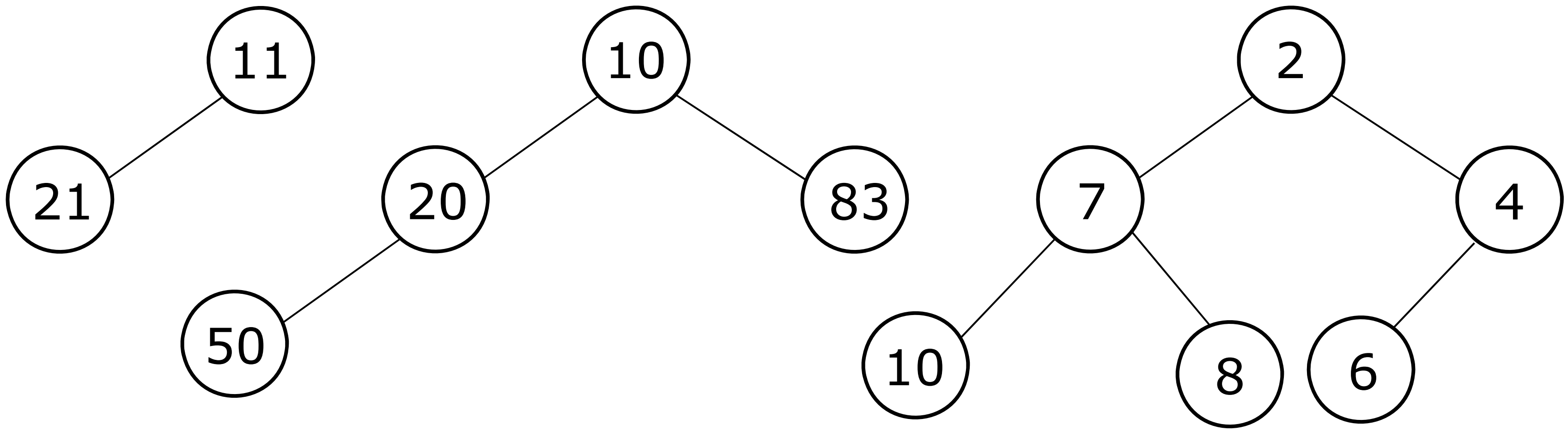
7.6.2 Definition of a Heap

- Max heap
  - A complete binary tree
  - The key value in each node is ***no smaller*** than the key values of its child nodes



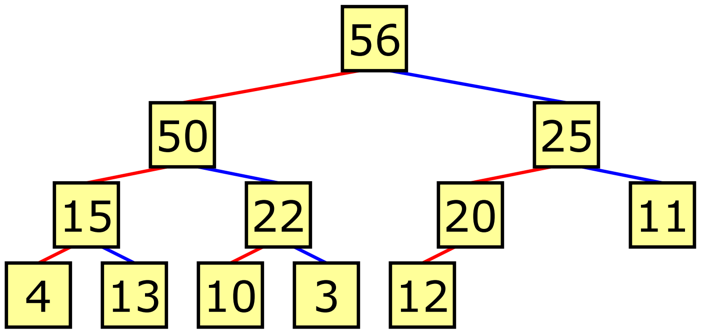
7.6.2 Definition of a Heap

- Min heap
  - A complete binary tree
  - The key value in each node is ***no larger*** than the key values of its child nodes



7.6.2 Definition of a Heap

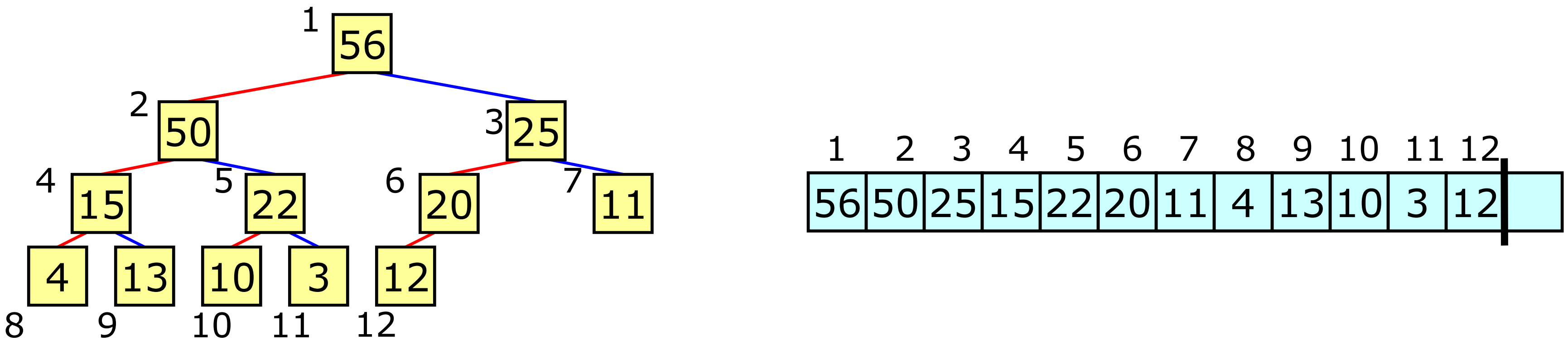
- Implementation of a heap
  - Implementation of a complete binary tree
    - Pointer-based
    - Array-based



56	50	25	15	22	20	11	4	13	10	3	12	

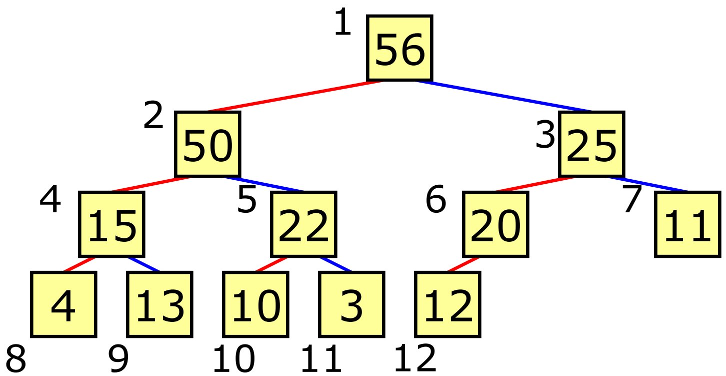
7.6.2 Definition of a Heap

- Implementation of a heap
  - Index the nodes of a heap from top to down, from left to right
  - Index the elements of an array from 1



7.6.2 Definition of a Heap

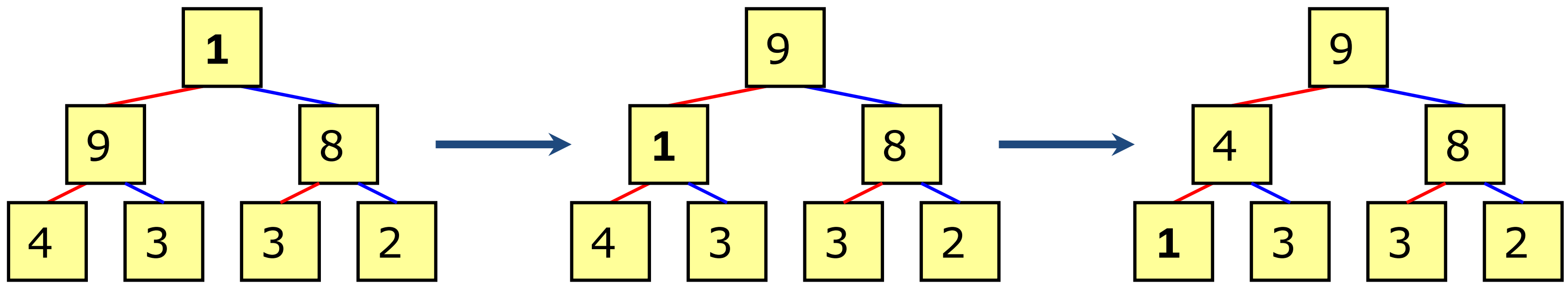
- Implementation of a heap
  - Parent of node k: k/2
  - Left child of node k: 2*k
  - Right child of node k: 2*k + 1



1	2	3	4	5	6	7	8	9	10	11	12	
56	50	25	15	22	20	11	4	13	10	3	12	

7.6.2 Definition of a Heap

- Heapify (k)
  - From node k, reorganize a tree to a heap
  - Topdown heapify ( )
    - From root node to leaf node, build a heap
  - Bottomup heapify ( )
    - From leaf node to root node, build a heap
  - Ex) Topdown heapify ( )



7.6.3 Insertion into a Heap

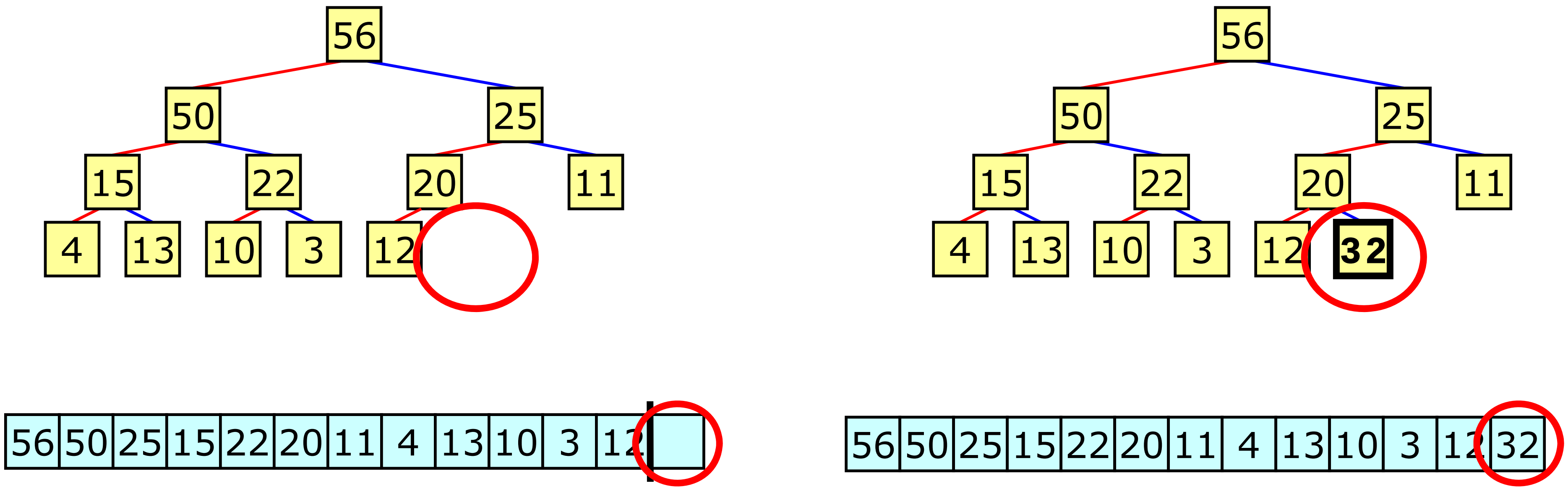
- Insert an element to a max heap

  (1) Insert an element to the last position of the heap (no longer heap)

  (2) Using heapify ( ), reorganize the newly inserted heap to a heap

7.6.3 Insertion into a Heap

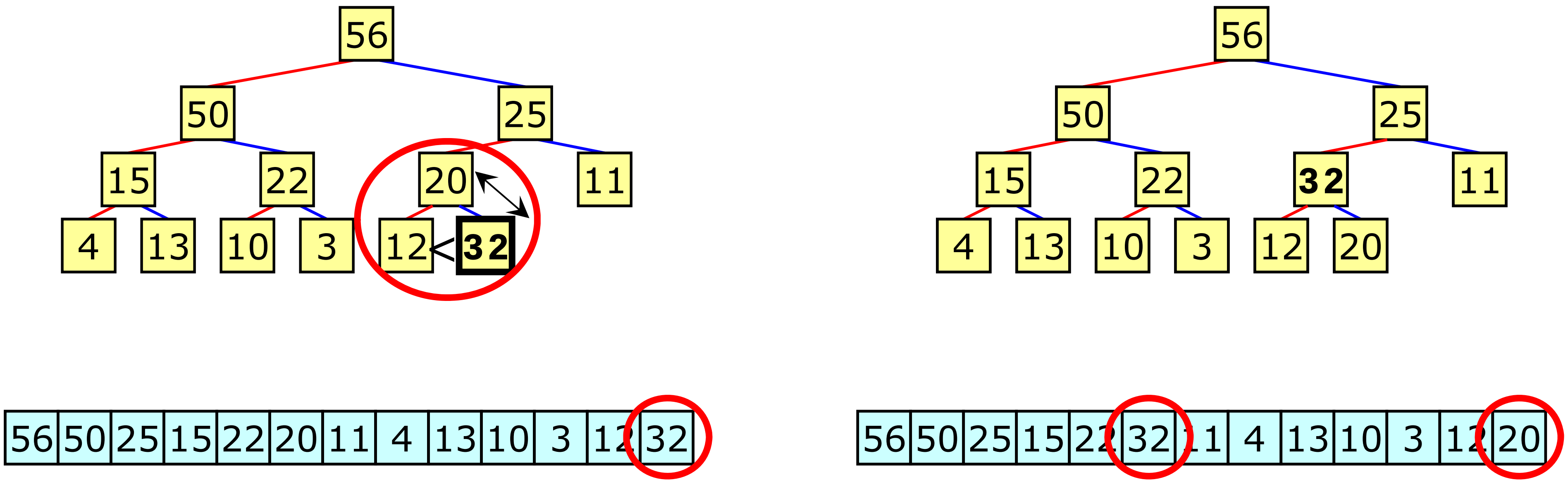
- Insert an element to a max heap
  - (1) Insert an element to the last position of the heap (no longer heap)



7.6.3 Insertion into a Heap

- Insert an element to a max heap

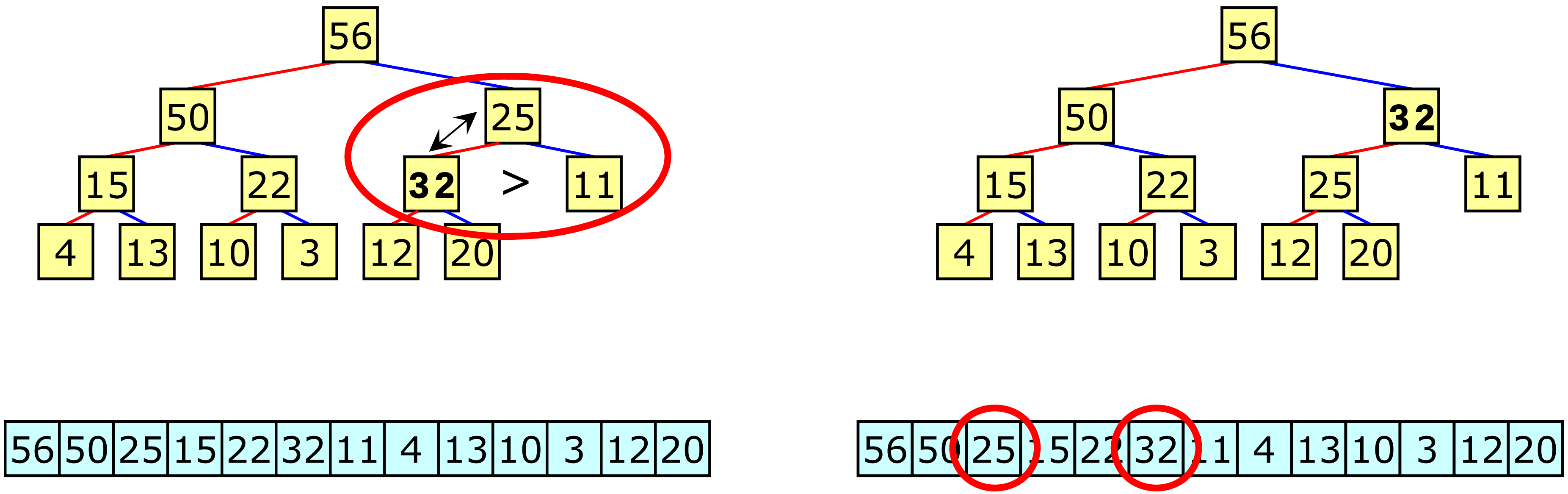
  (2) Using heapify ( ), reorganize the newly inserted heap to a heap



7.6.3 Insertion into a Heap

- Insert an element to a max heap

  (2) Using heapify ( ), reorganize the newly inserted heap to a heap



7.6.3 Insertion into a Heap

- Time complexity of push ( )
  - Heap → complete binary tree of n nodes
  - Height of heap → log (n)

  - Time complexity for push ( )
    → O(log (n))

7.6.3 Insertion into a Heap

- Exercise
  - Build a max heap by inserting the following values:

7, 16, 49, 82, 5, 31, 6, 2, 44

7.6.3 Insertion into a Heap

- Exercise
  - Build a min heap by inserting the following values:

7, 16, 49, 82, 5, 31, 6, 2, 44

7.6.4 Deletion from a Heap

- Delete from a max heap

  (1) Remove the root of heap and return the element of the root node

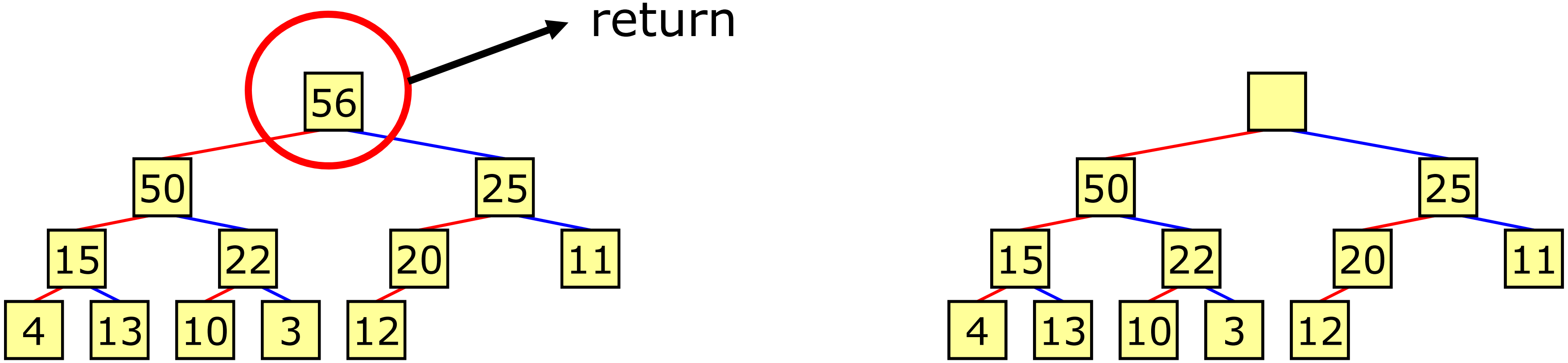
  (2) Move the element of the last node to the root node and remove the last node

  (3) Apply Heapify ( ) to maintain the heap

7.6.4 Deletion from a Heap

- Delete from a max heap

  (1) Remove the root of heap and return the element of the root node



7.6.4 Deletion from a Heap

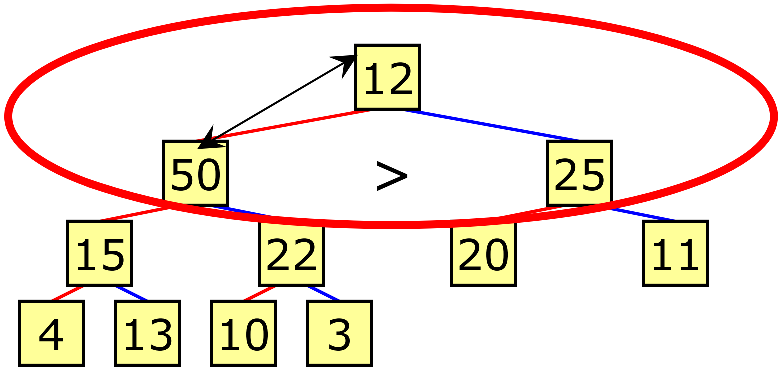
- Delete from a max heap

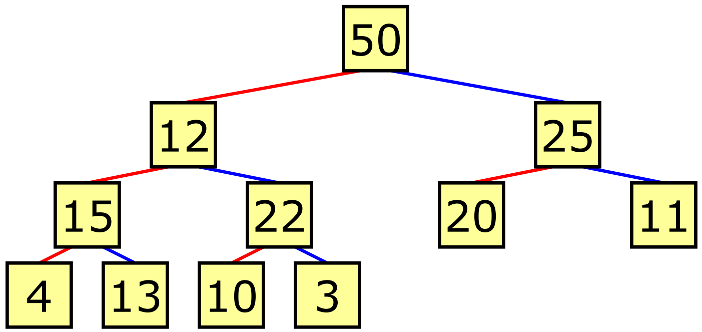
  (2) Move the element of the last node to the root node and remove the last node

7.6.4 Deletion from a Heap

- Delete from a max heap

  (3) Apply heapify ( ) to maintain the structure of max heap





7.6.4 Deletion from a Heap

- Delete from a max heap
  - (3) Apply heapify ( ) to maintain the structure of max heap

7.6.4 Deletion from a Heap

- Time complexity of a pop ( )
  - Heap → complete binary tree of n nodes
  - Height of heap → log (n)

  - Time complexity for pop ( )
    → O(log (n))

7.6.5 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		***O(log n)***	***O(log n)***	***O(n)***	***O(1)***	***O(log n)***
Hash table		**O(1)**	**O(1)**	**O(1)**	**O(1)**	**O(1)**

Contents