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

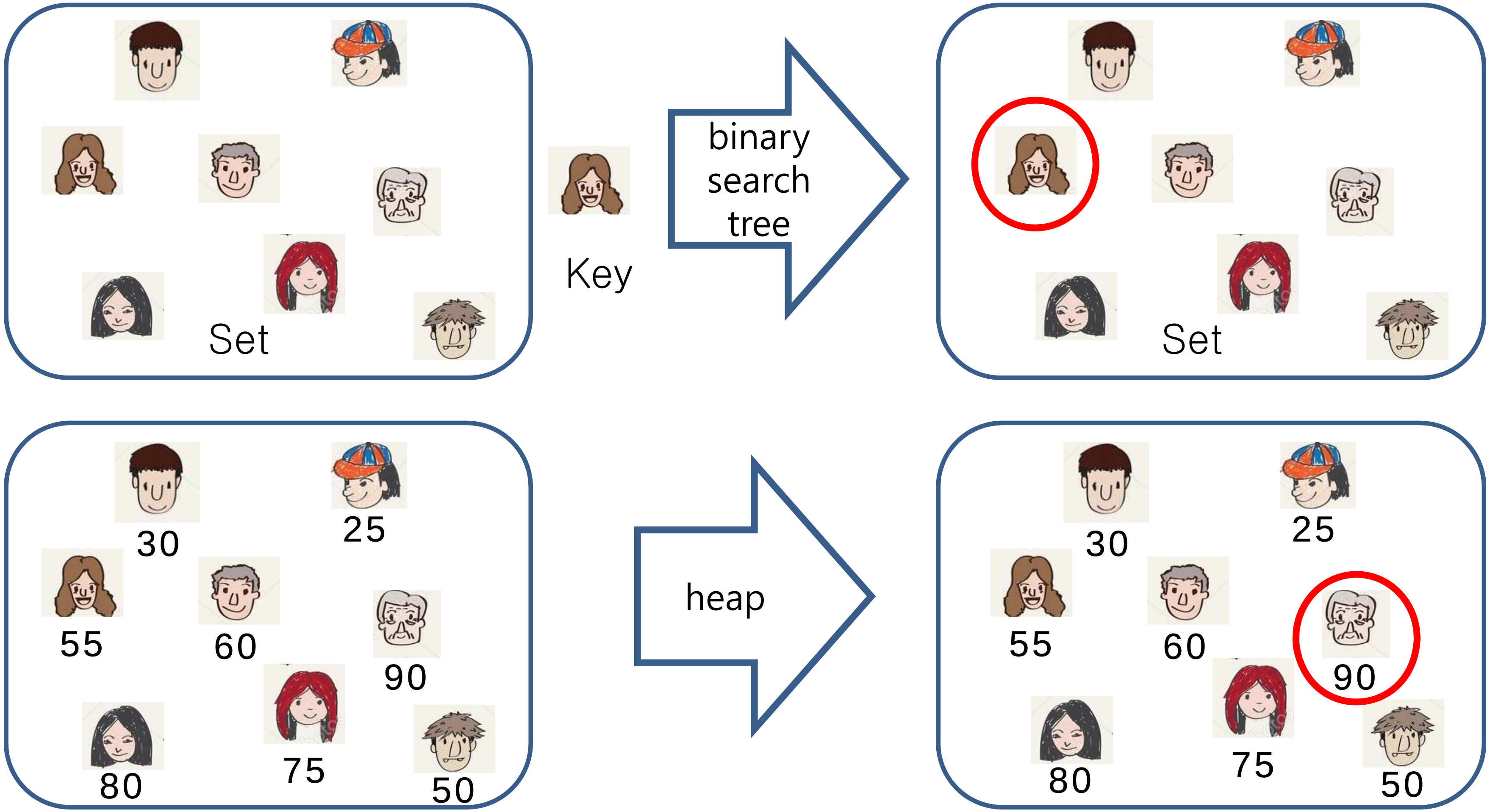
7.5.5 Time complexity on a binary search tree

7.5 Binary search tree

- 자료구조
  - Data를 효율적으로 **관리**하는 기법
  - 관리
    - 삽입, 삭제, **탐색**, ... Etc
  - 탐색 (search)
    - 임의의 원소 찾기
    - 가장 늦게/먼저 온 원소 찾기 (stack/queue)
    - 1등 찾기
  - "임의의 원소 찾기"의 요구 조건: set + key
    - 김서방 (key) 을 찾아라 (X)
    - 서울 (set) 에서 김서방 (key)을 찾아라 (O)

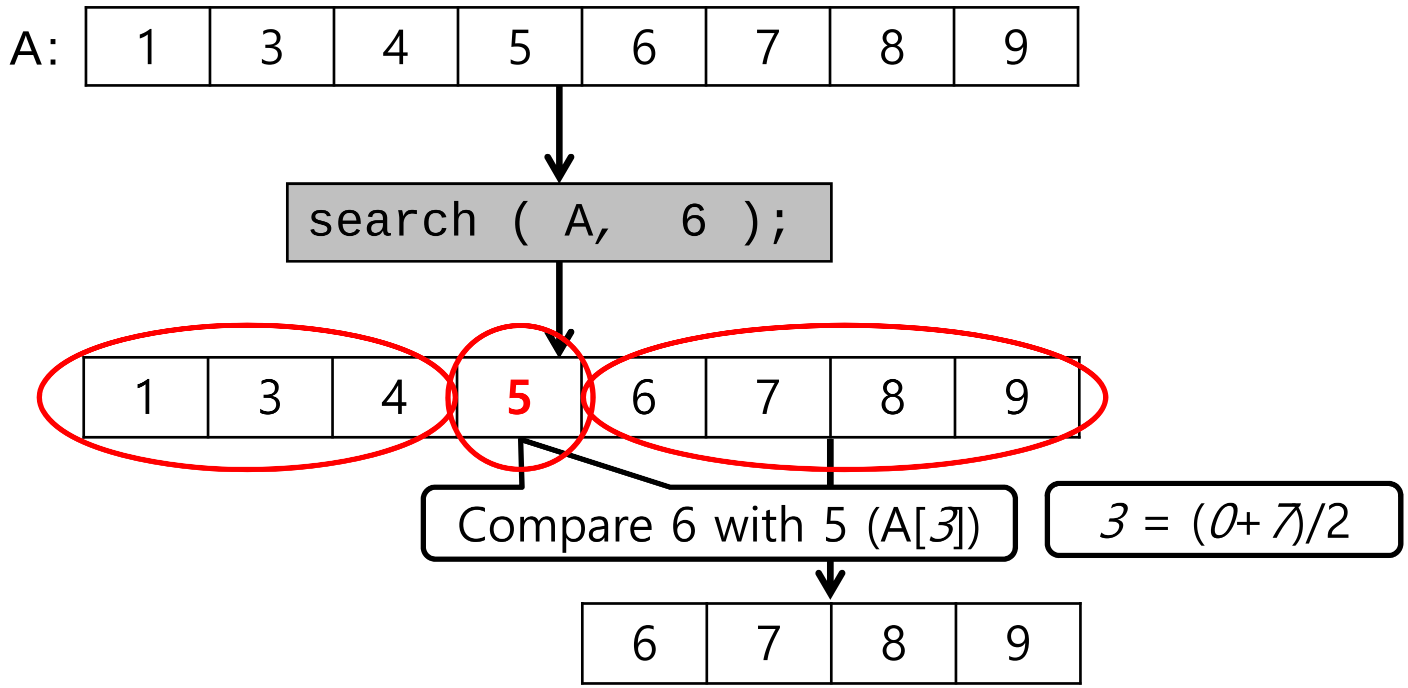
7.5 Binary search tree

- 탐색: 임의의 원소 찾기 VS 1등 찾기



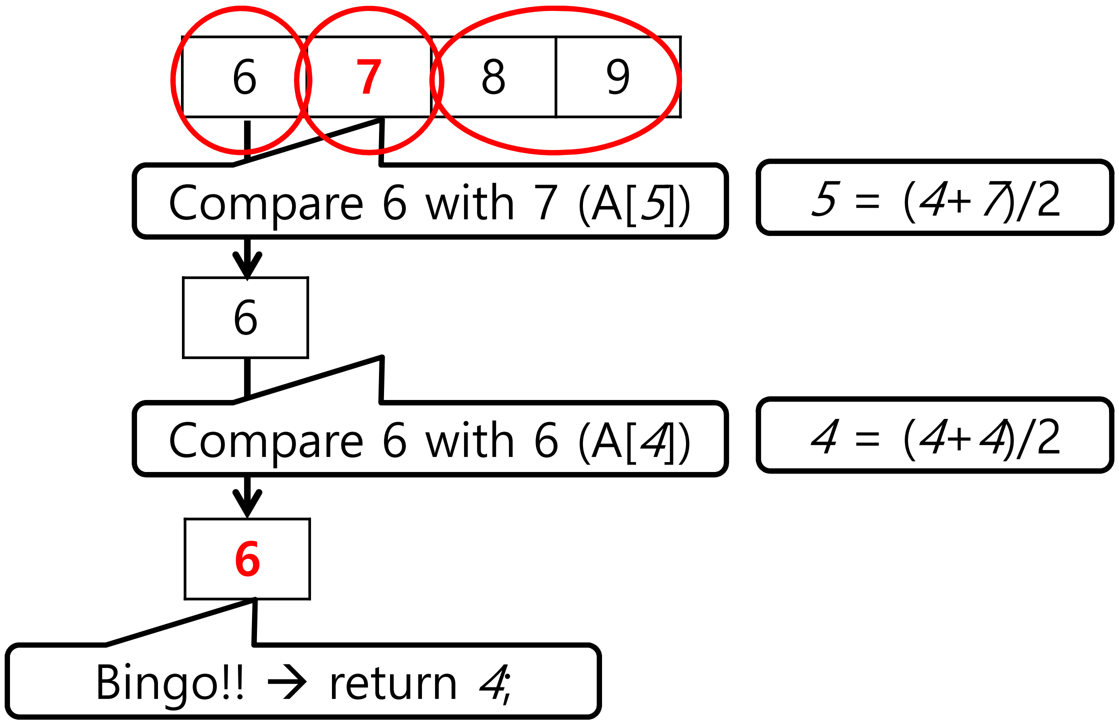
7.5.1 Definition

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



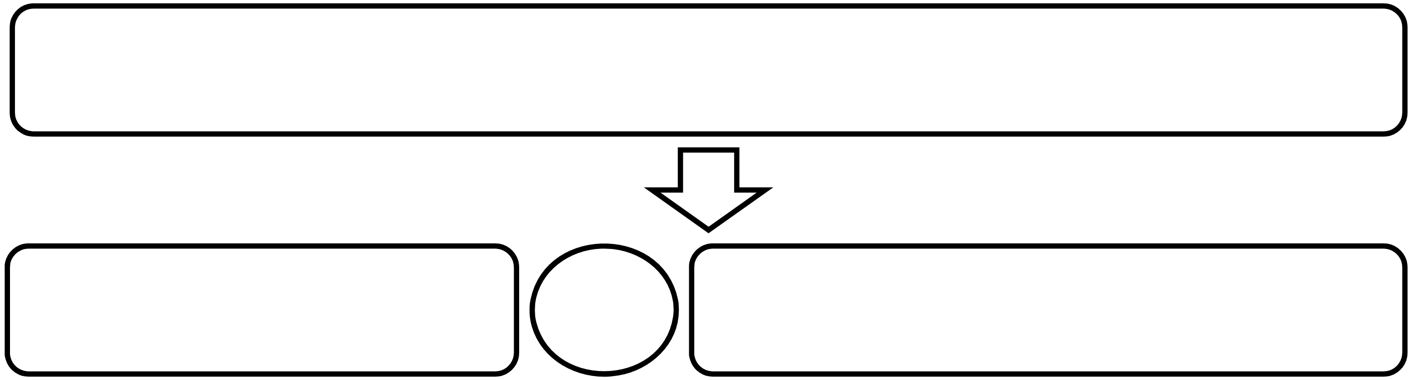
7.5.1 Definition

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



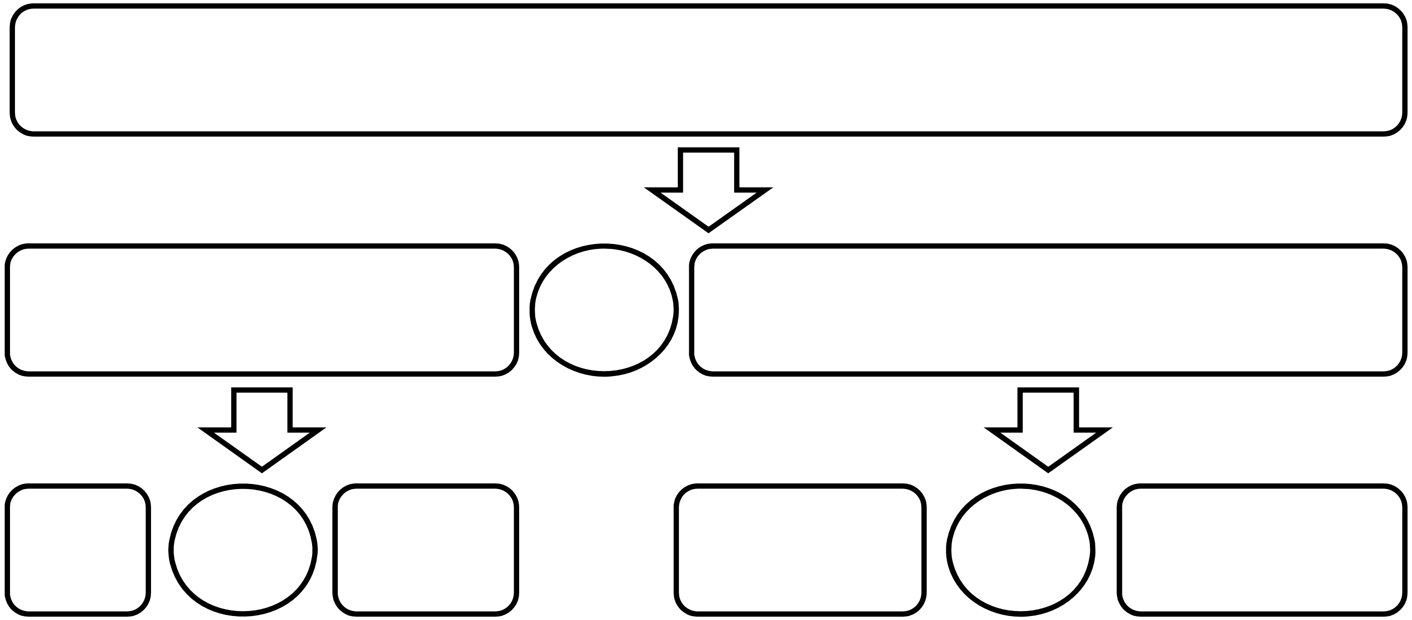
7.5.1 Definition

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



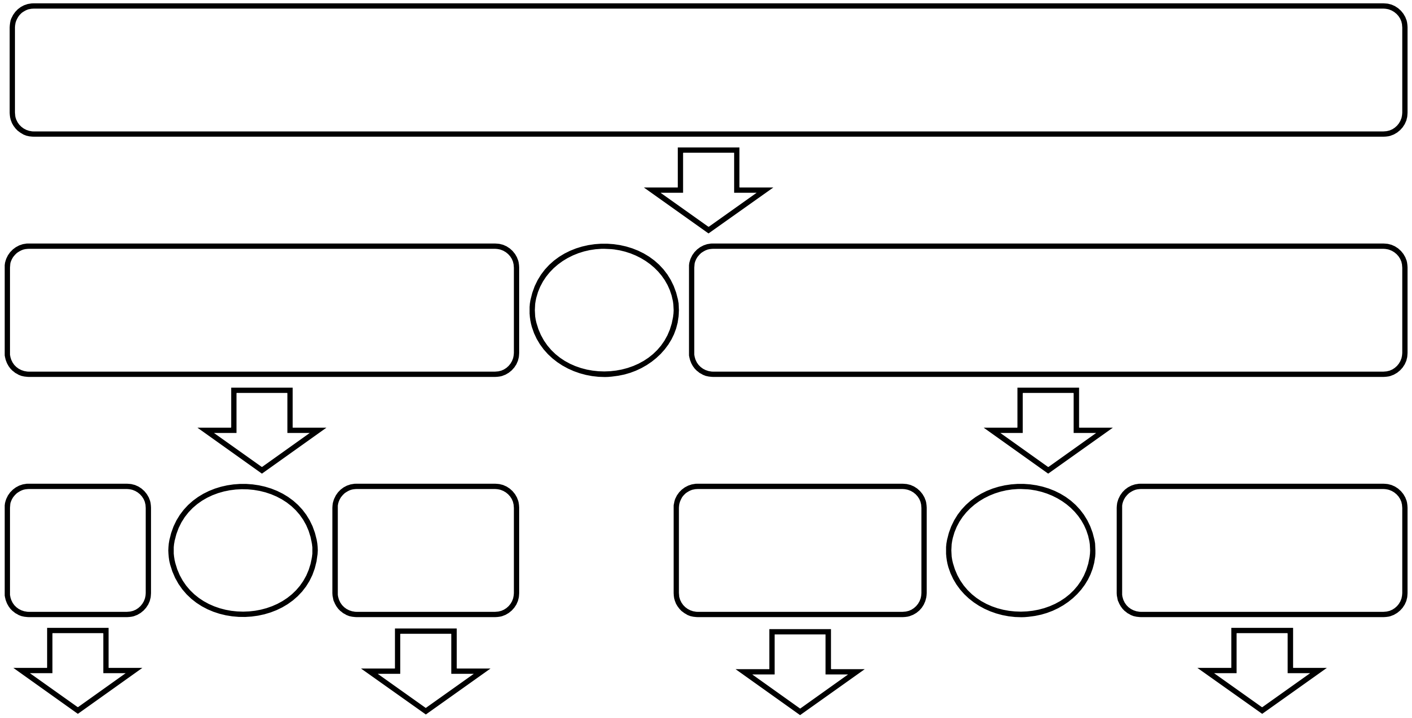
7.5.1 Definition

- 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**

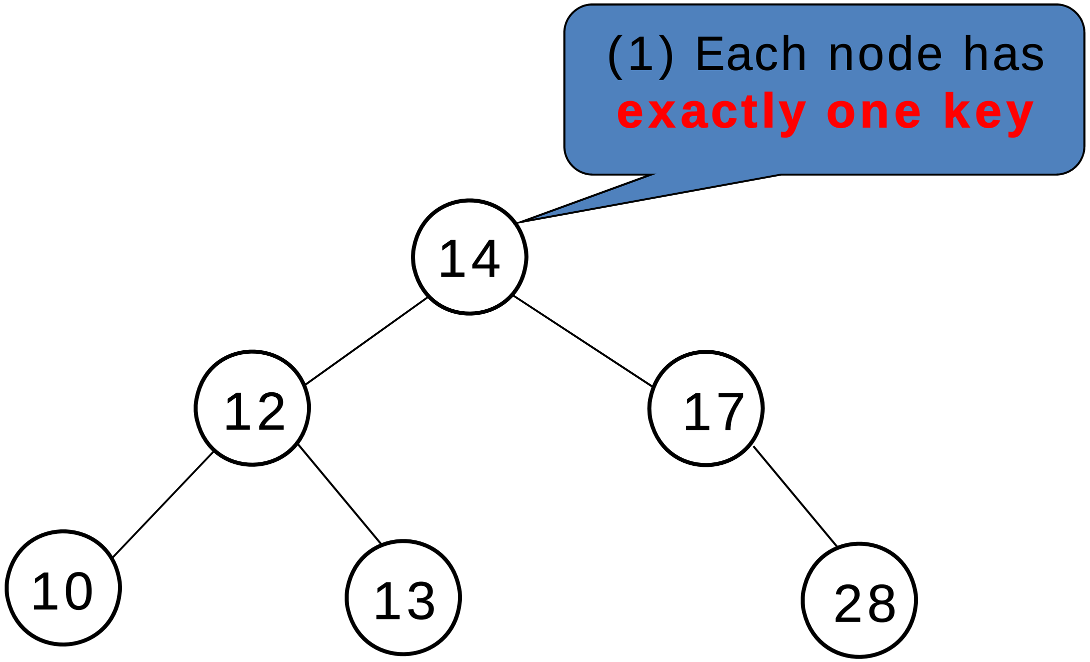
7.5.1 Definition

- 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						

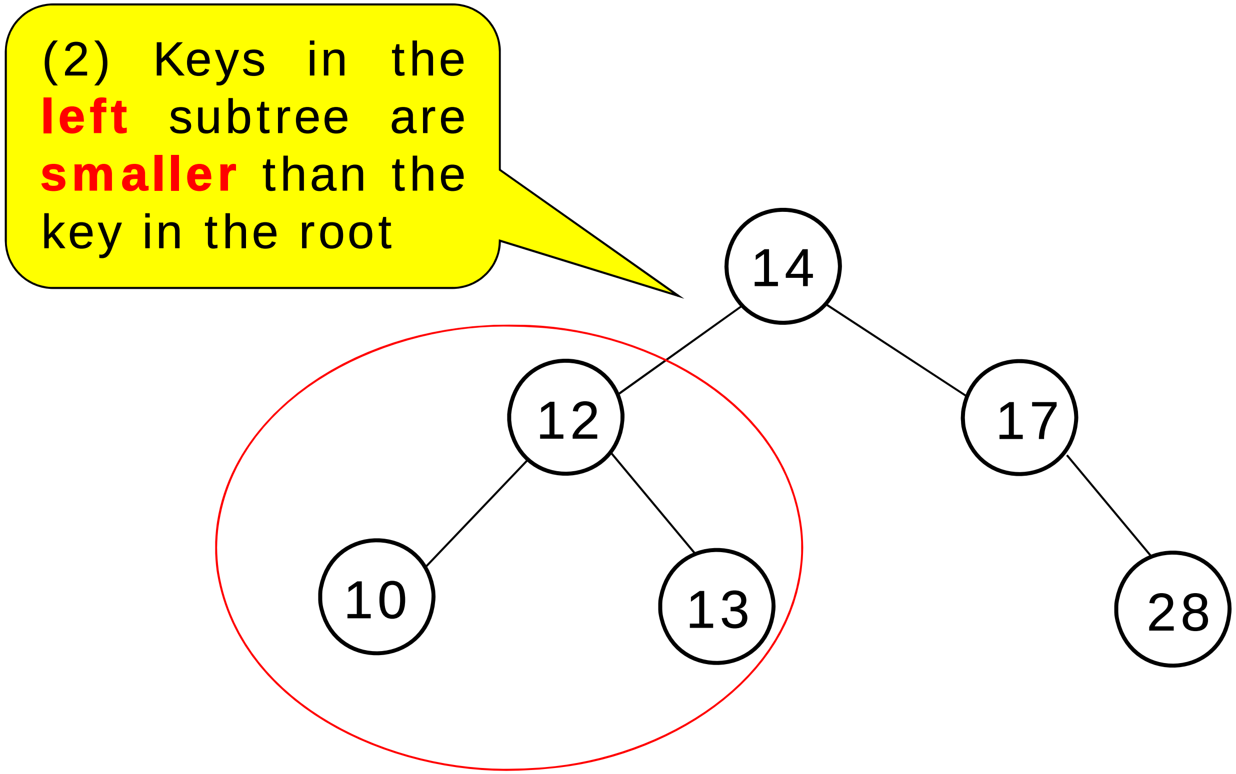
7.5.1 Definition

- Binary search tree



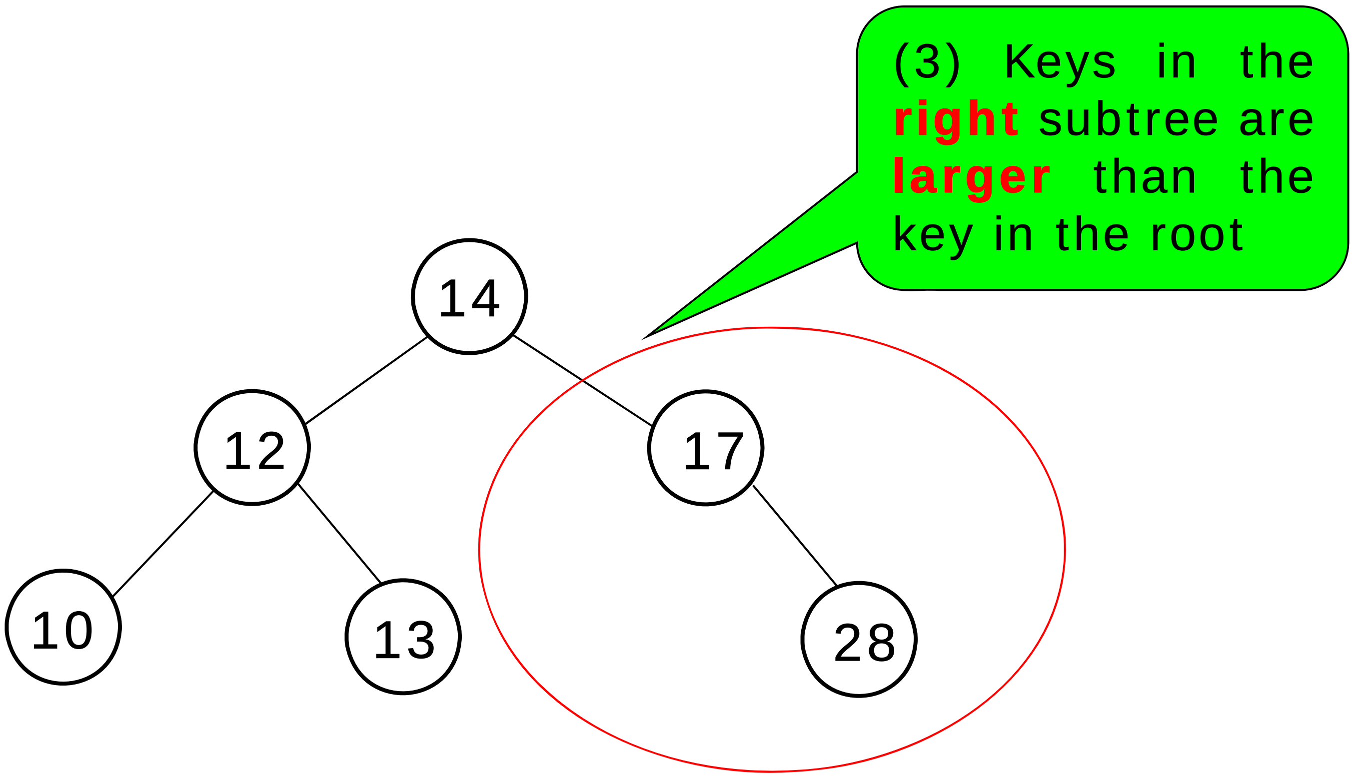
7.5.1 Definition

- Binary search tree



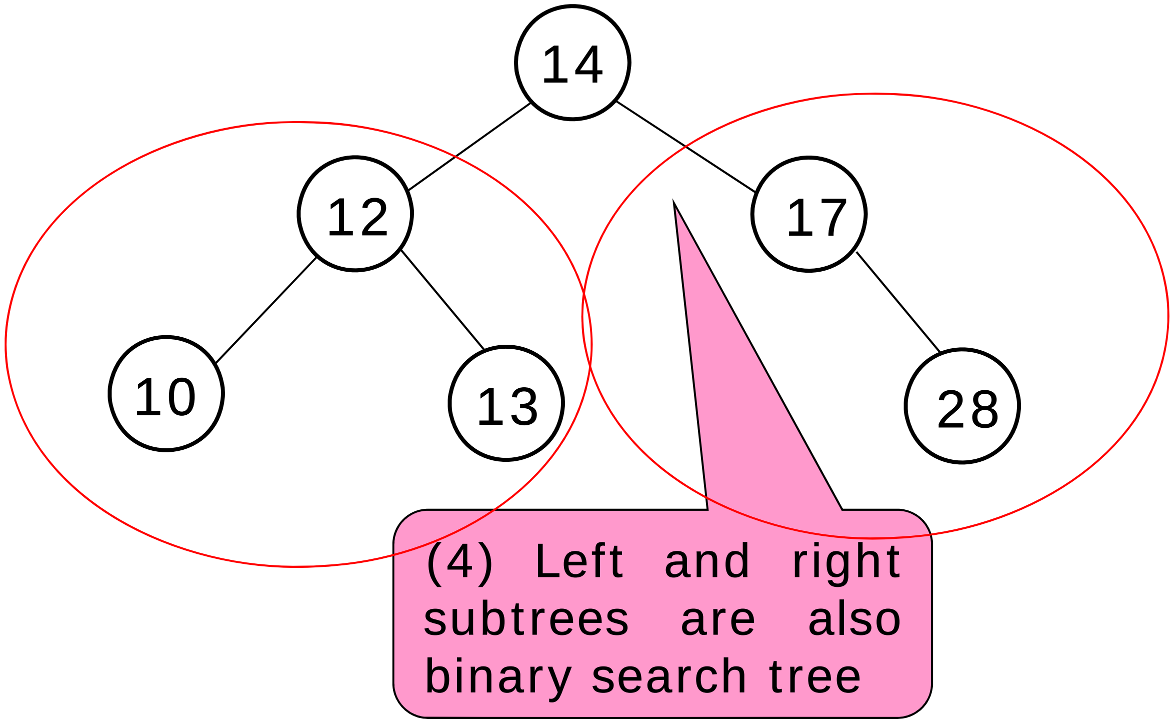
7.5.1 Definition

- Binary search tree



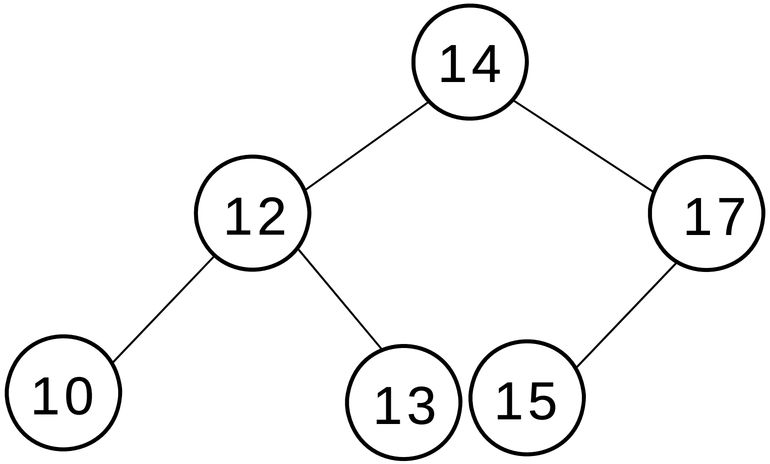
7.5.1 Definition

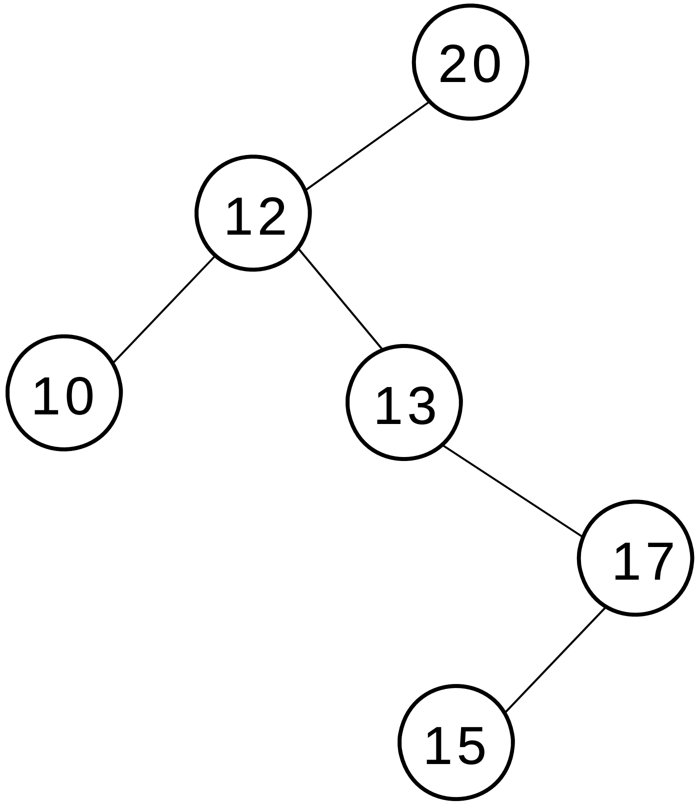
- Binary search tree



7.5.1 Definition

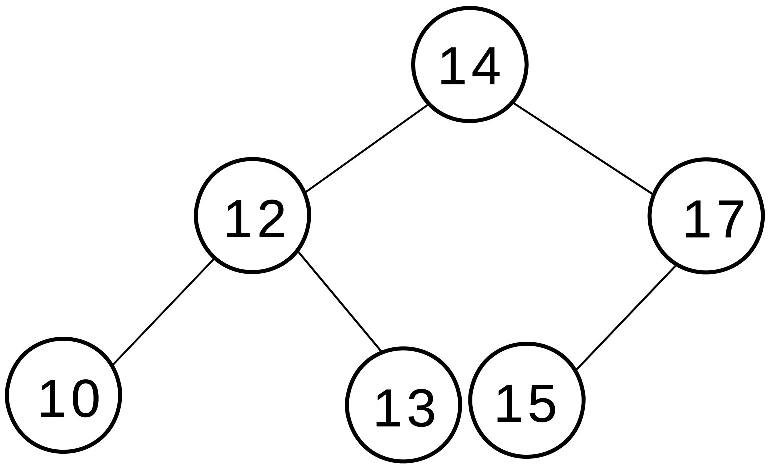
- Binary search trees





7.5.1 Definition

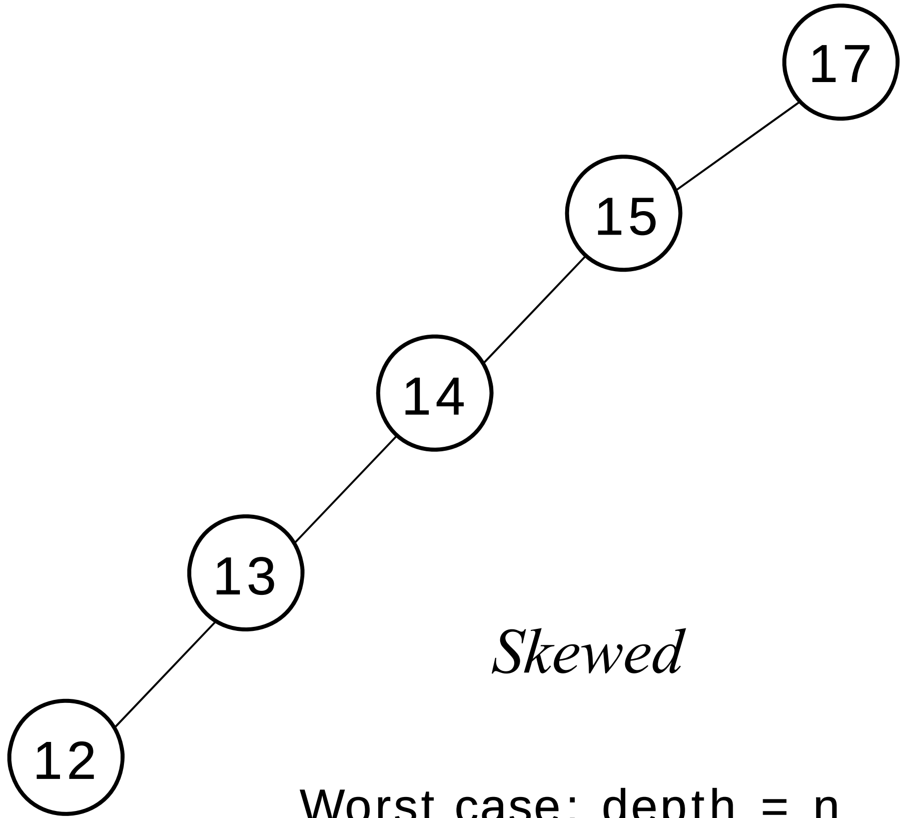
- Binary search trees (good and bad)



Balanced

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

Best case: depth = log n



Skewed

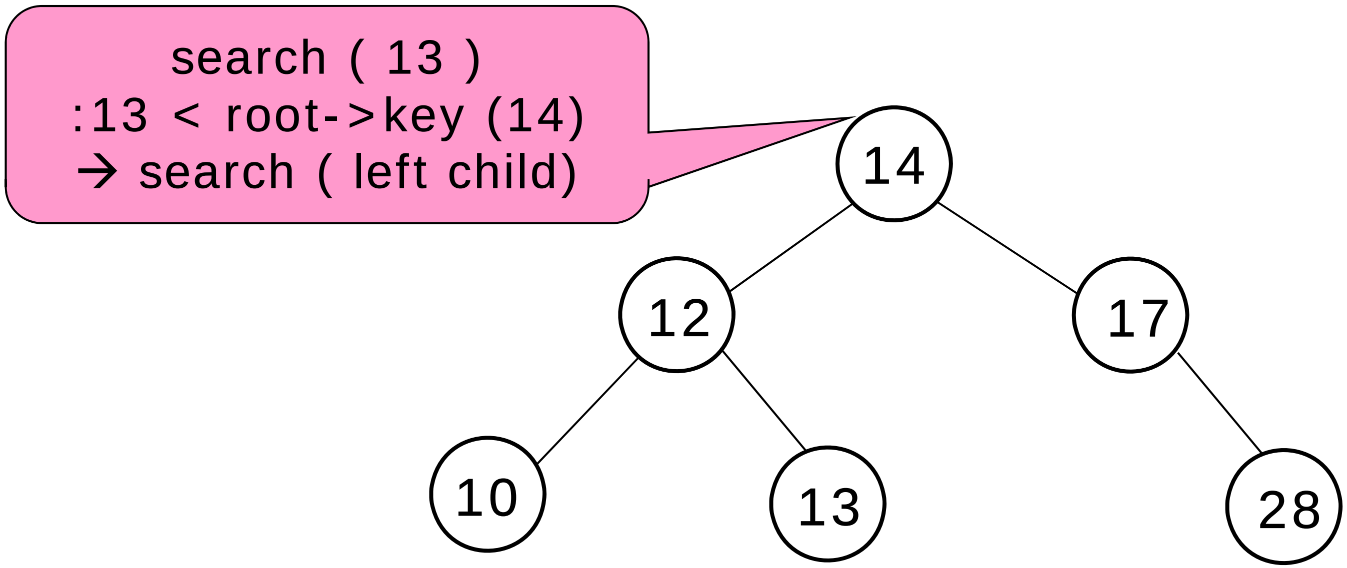
Worst case: depth = n

7.5.2 Search

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

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

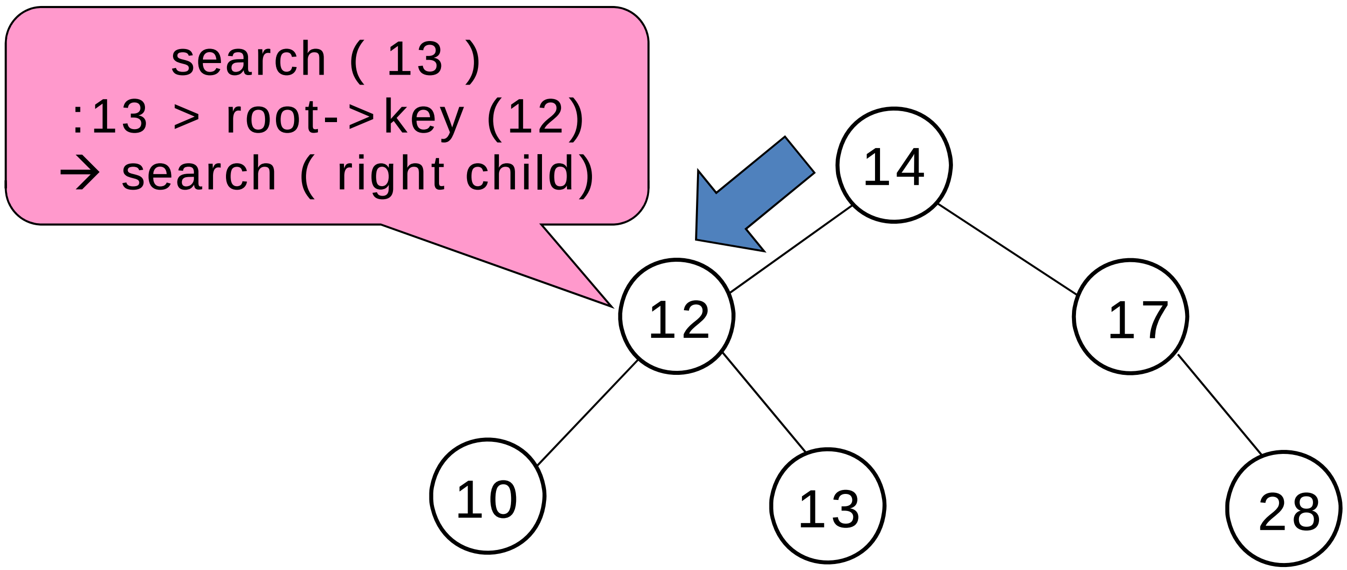
```
root->search ( 13 );
```



7.5.2 Search

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

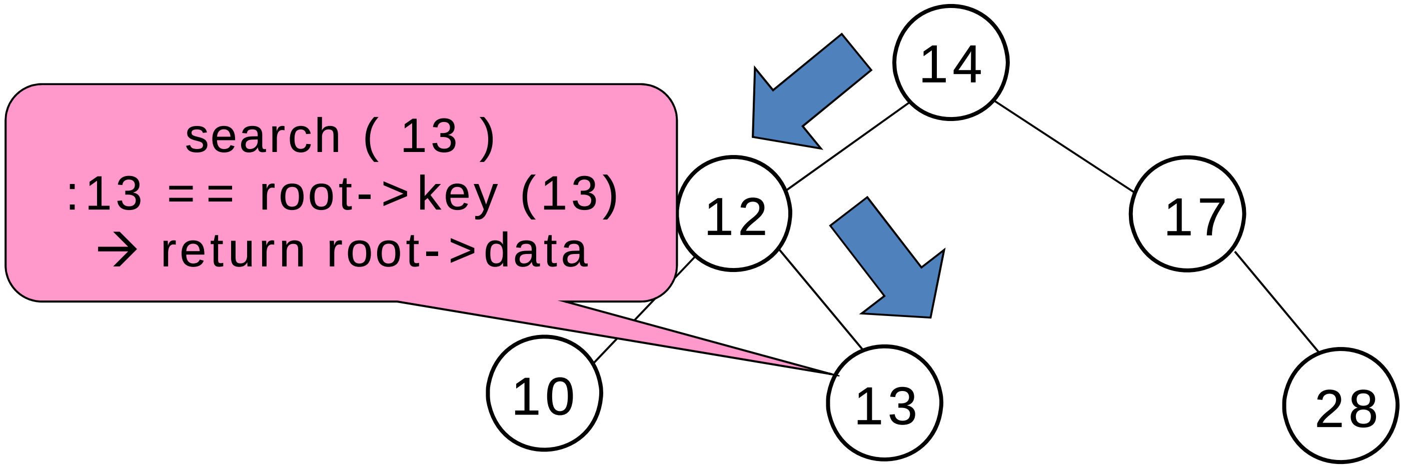
```
element node::search (KEY key )
```



7.5.2 Search

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

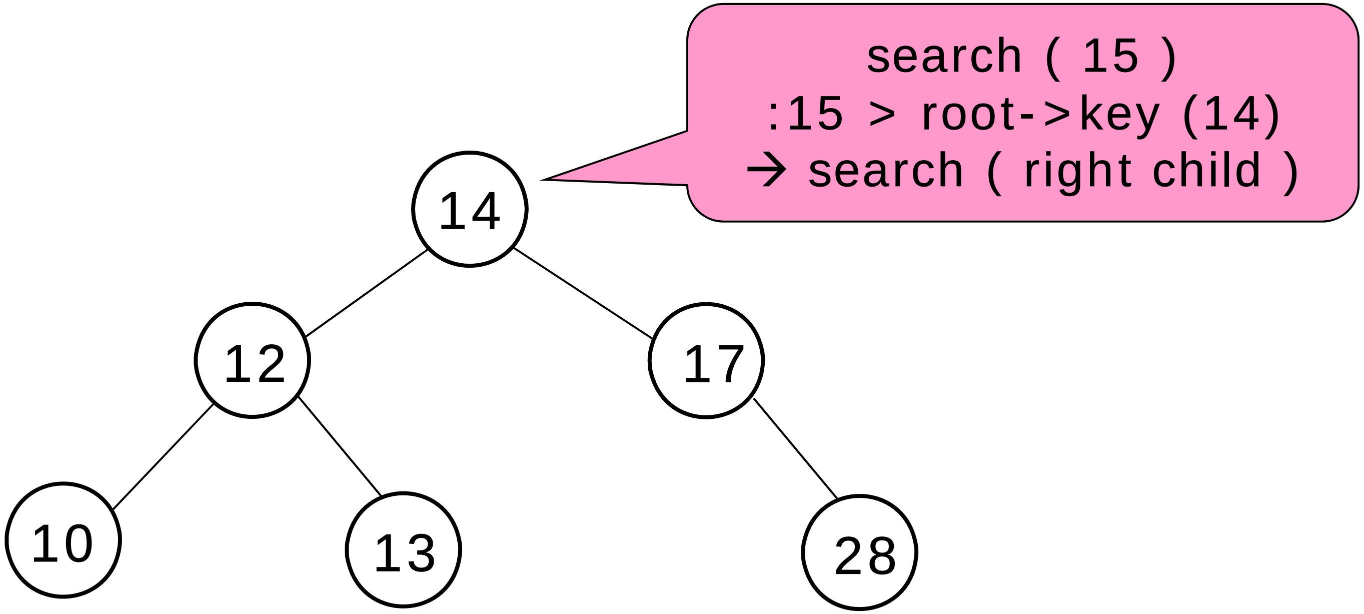
```
element node::search (KEY key )
```



7.5.2 Search

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

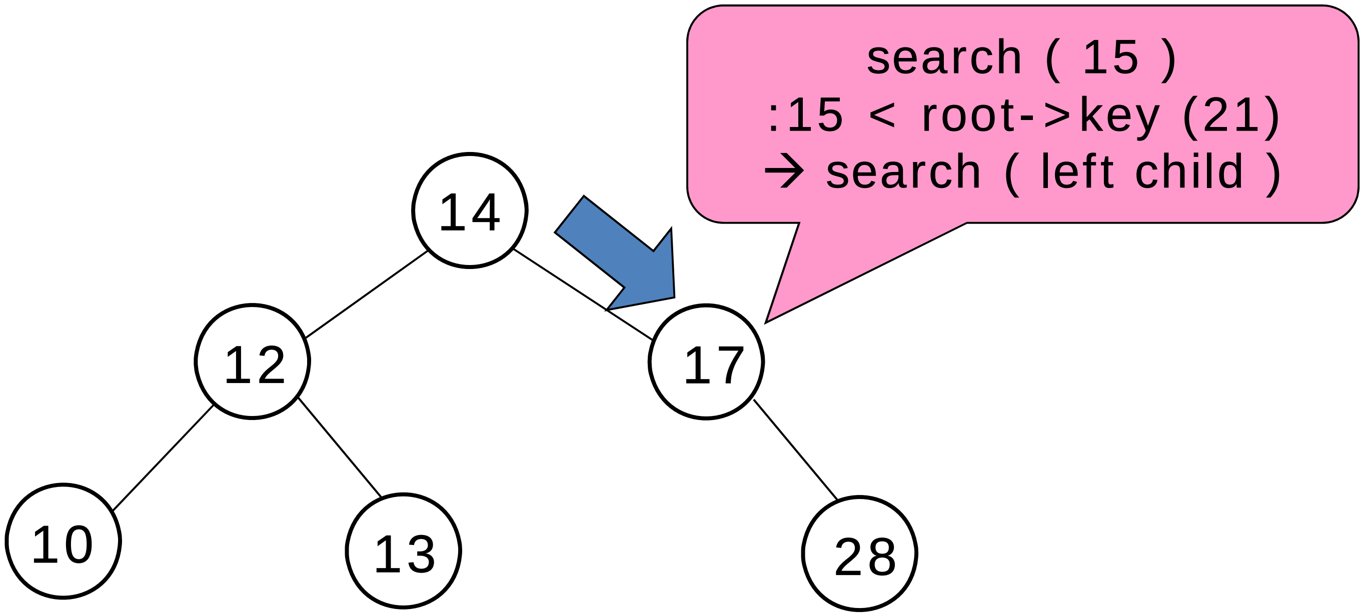
```
element node::search (KEY key )
```



7.5.2 Search

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

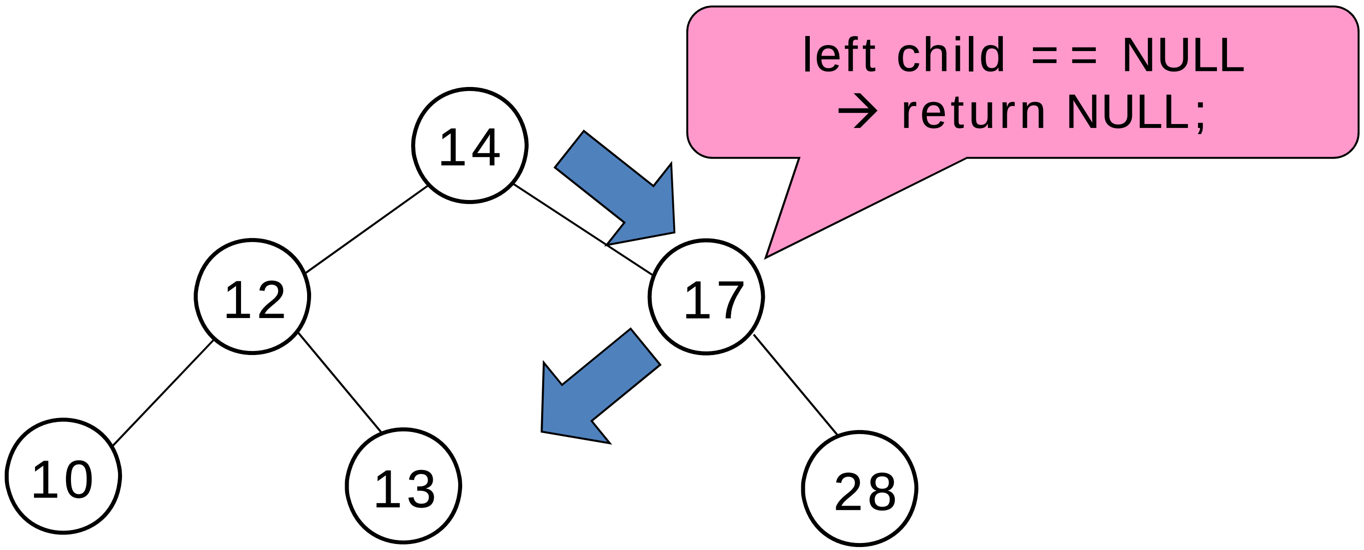
```
element node::search (KEY key )
```



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");
        }
}
```

7.5.2 Search

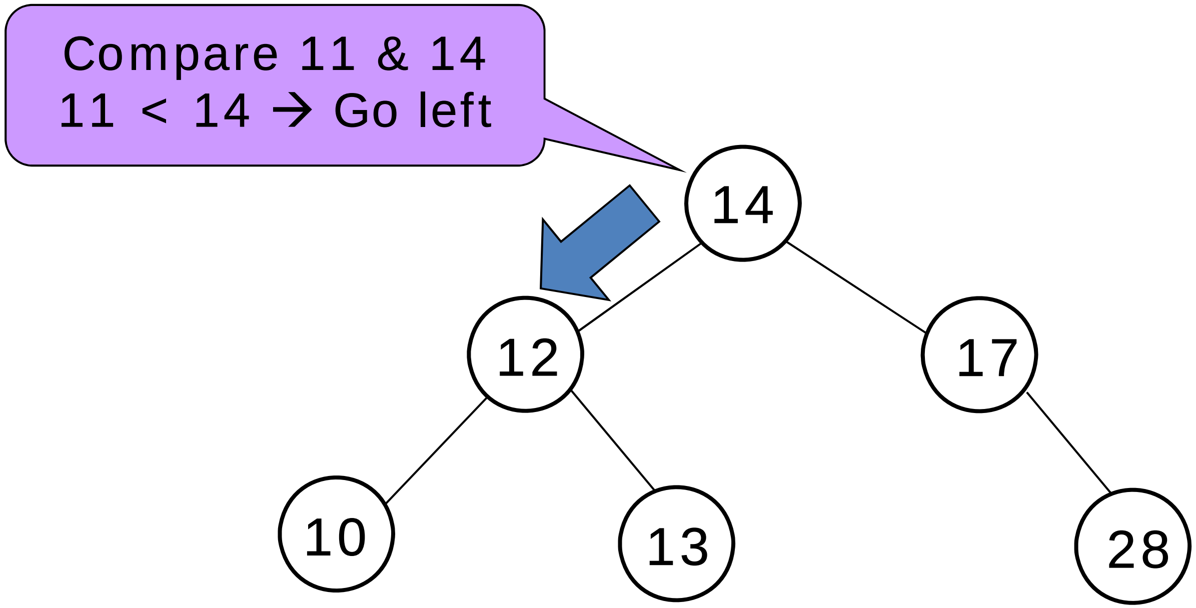
- Time complexity
  - Best case
    - A binary tree with n nodes has depth of log n.
    - At worst case, search ends at the leaf nodes.
    - So, the best case time complexity is O(log n).
  - Worst case
    - A binary tree with n nodes has depth of n.
    - At worst case, search ends at the leaf nodes.
    - So, the best case time complexity is O(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

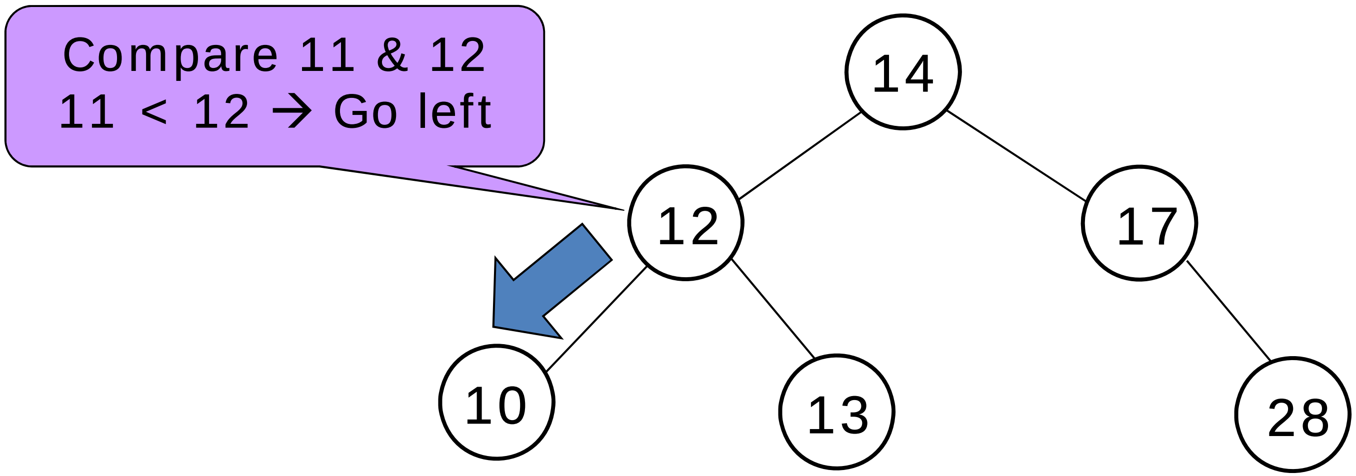
7.5.3 Insert

- Example
  - Insert <11>



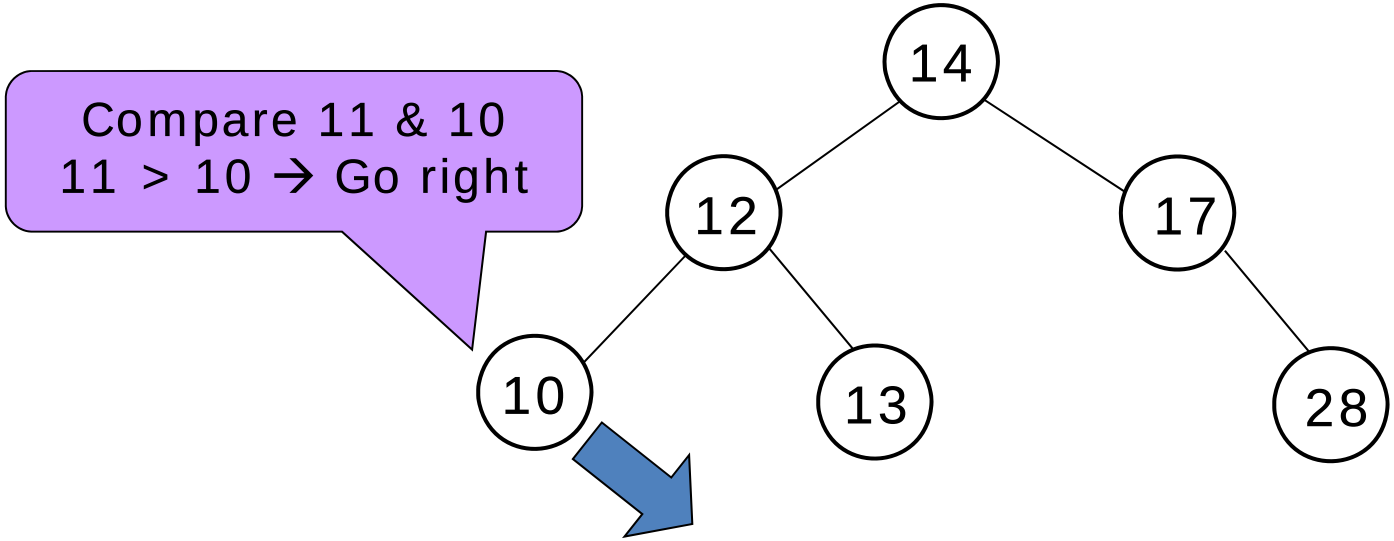
7.5.3 Insert

- Example
  - Insert <11>



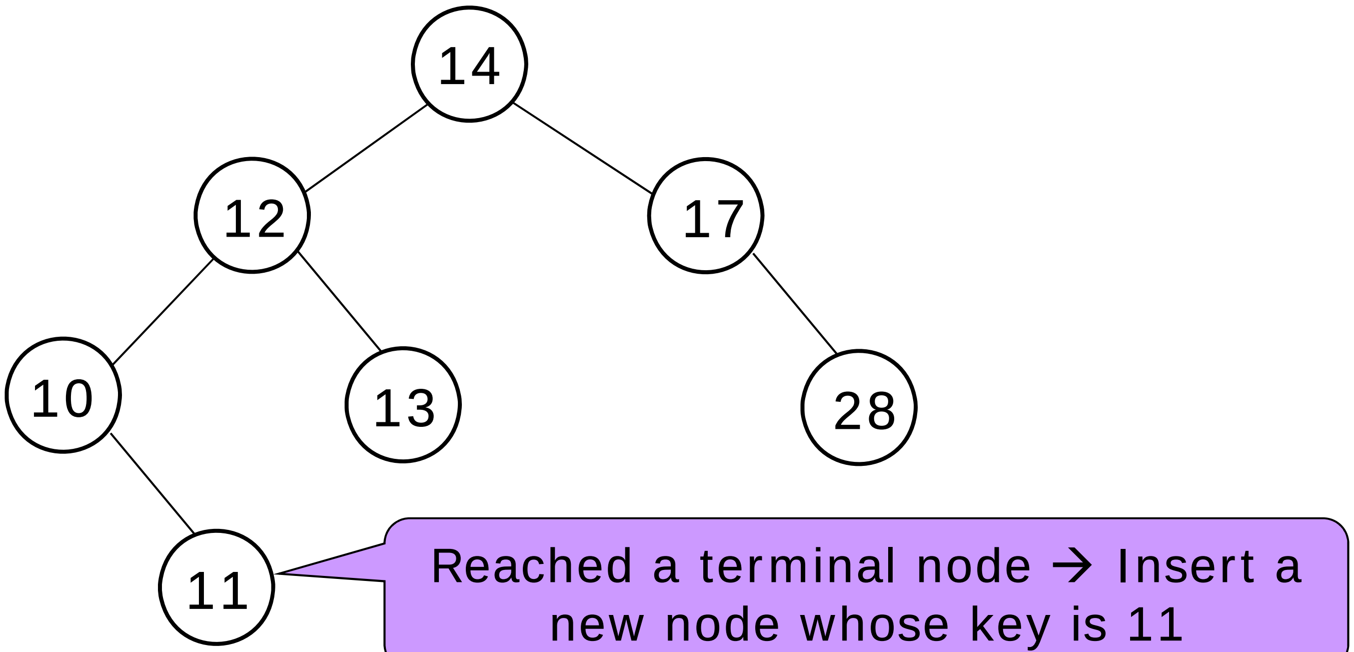
7.5.3 Insert

- Example
  - Insert <11>



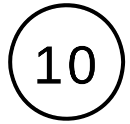
7.5.3 Insert

- Example
  - Insert <11>



7.5.3 Insert

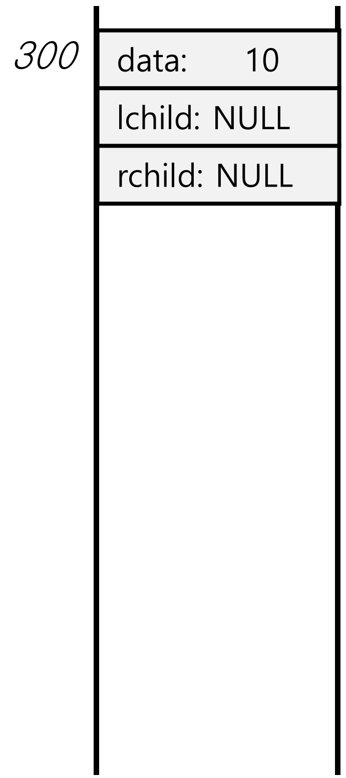
- Recursive implementation




```
void node::insert(int ndata)
{
//   degenerate case: root node에 삽입
     if (this->key == -1) {
          this->key = ndata;
          return;
     }
//   key와 같으면
     if ( this->key == ndata ) {
          printf("No duplicate data\n");
     }
//   key보다 크면
     else if (this->key < ndata ) {
          this->rchild->insert(ndata);
     }
//   key보다 작으면
     else {
          this->lchild->insert(ndata);
      }
}
```

Where to insert ?

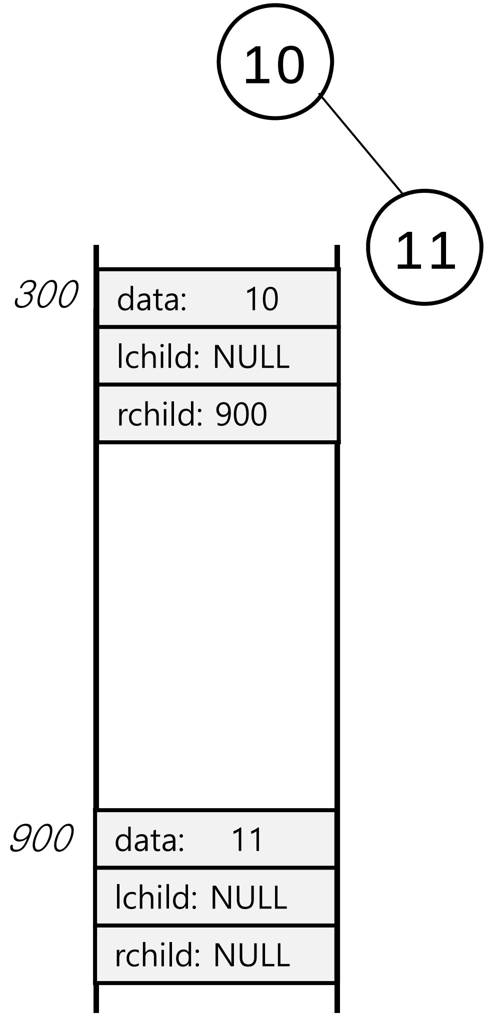
Where to insert ?



7.5.3 Insert

- Recursive implementation

```
//   key보다 크면
     if (this->key < ndata) {
         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;
         }
     }
//   key보다 작으면
     else {
         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;
         }
     }
```



7.5.3 Insert

- Time complexity
  - Best case
    - A binary tree with n nodes has depth of log n.
    - Insert ends at the leaf nodes.
    - So, the best case time complexity is O(log n).
  - Worst case
    - A binary tree with n nodes has depth of n.
    - Insert ends at the leaf nodes.
    - So, the best case time complexity is O(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

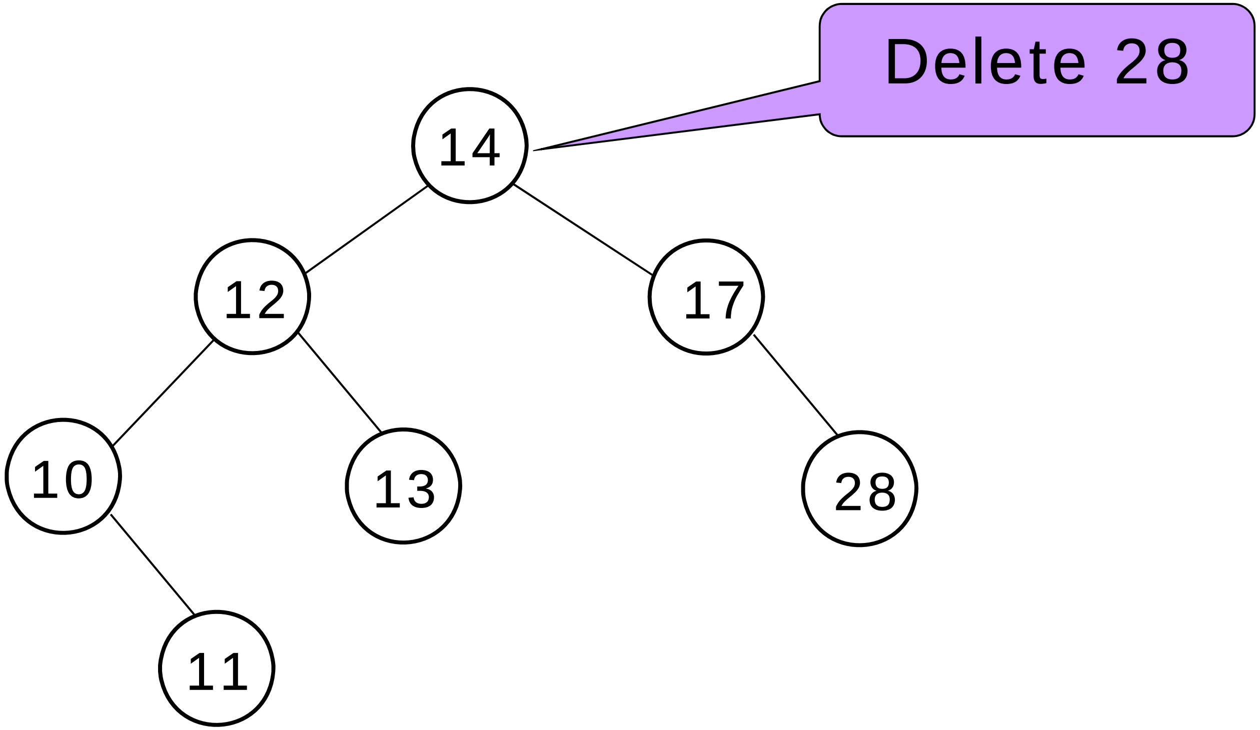
7.5.4 Delete

- Deleting leaf nodes

  → Delete the node

7.5.4 Delete

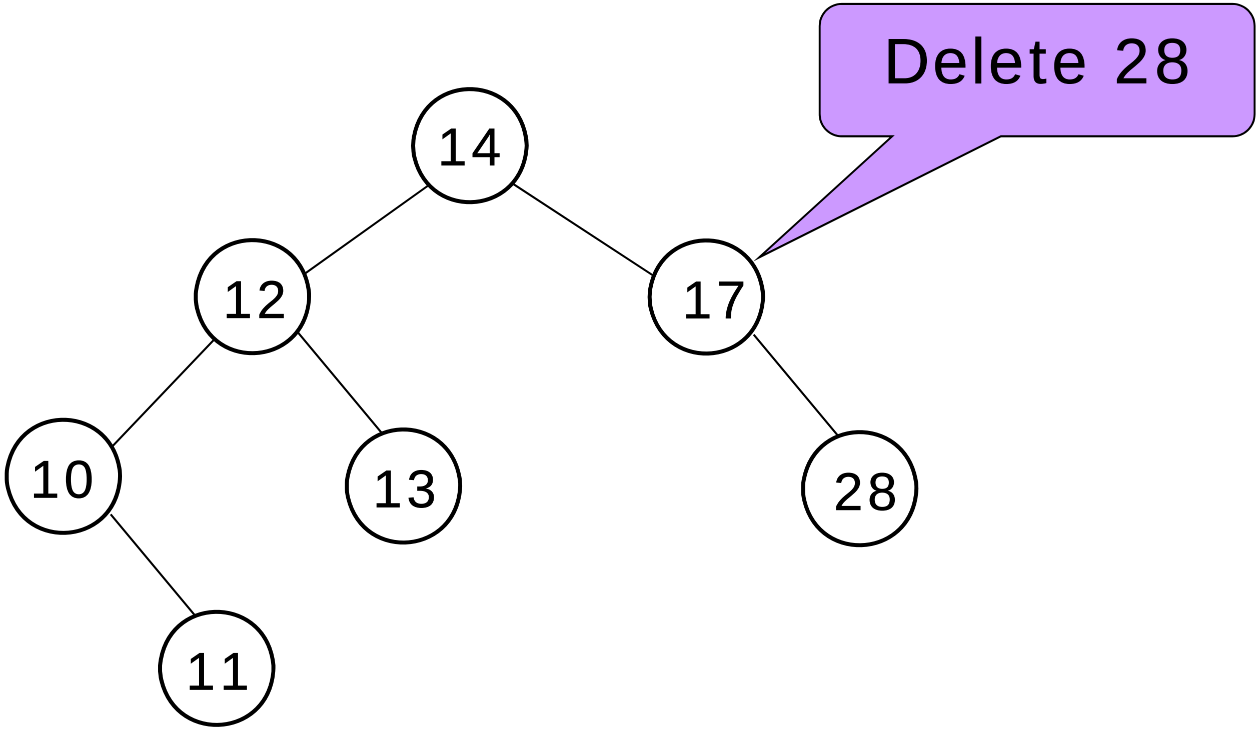
- Deleting leaf nodes
  → Delete the node



7.5.4 Delete

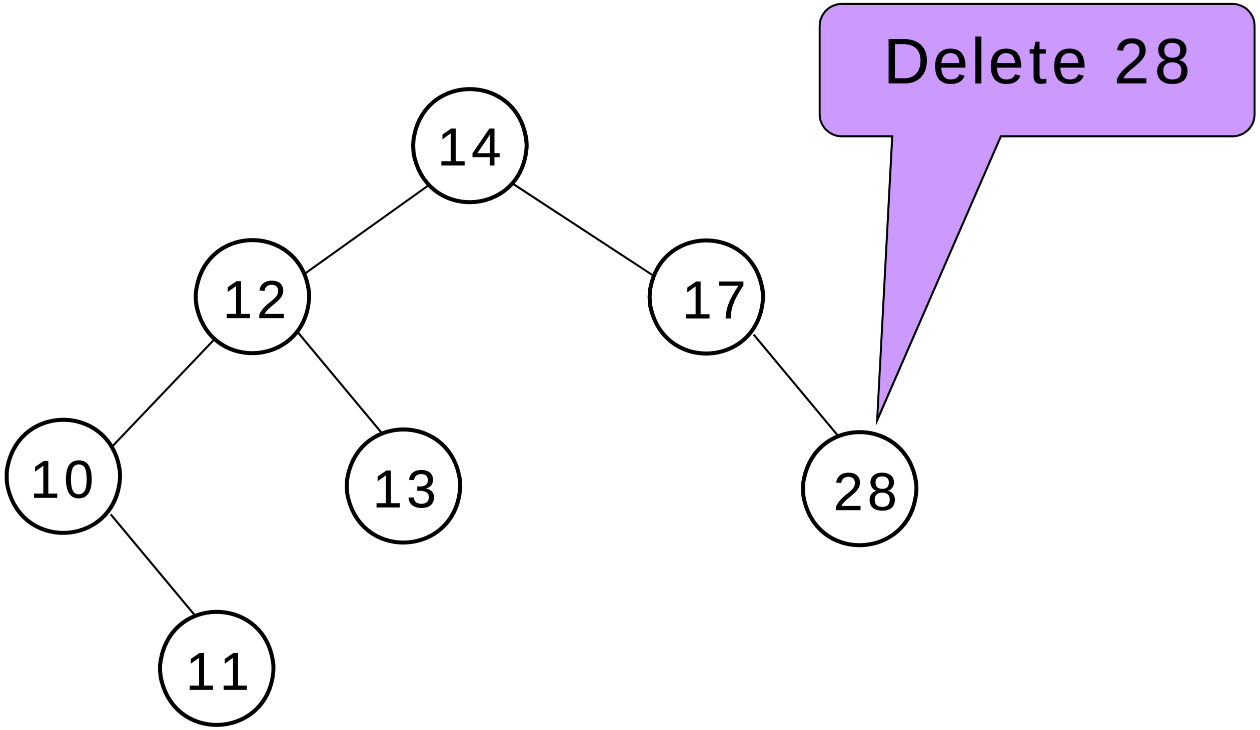
- Deleting leaf nodes

→ Delete the node



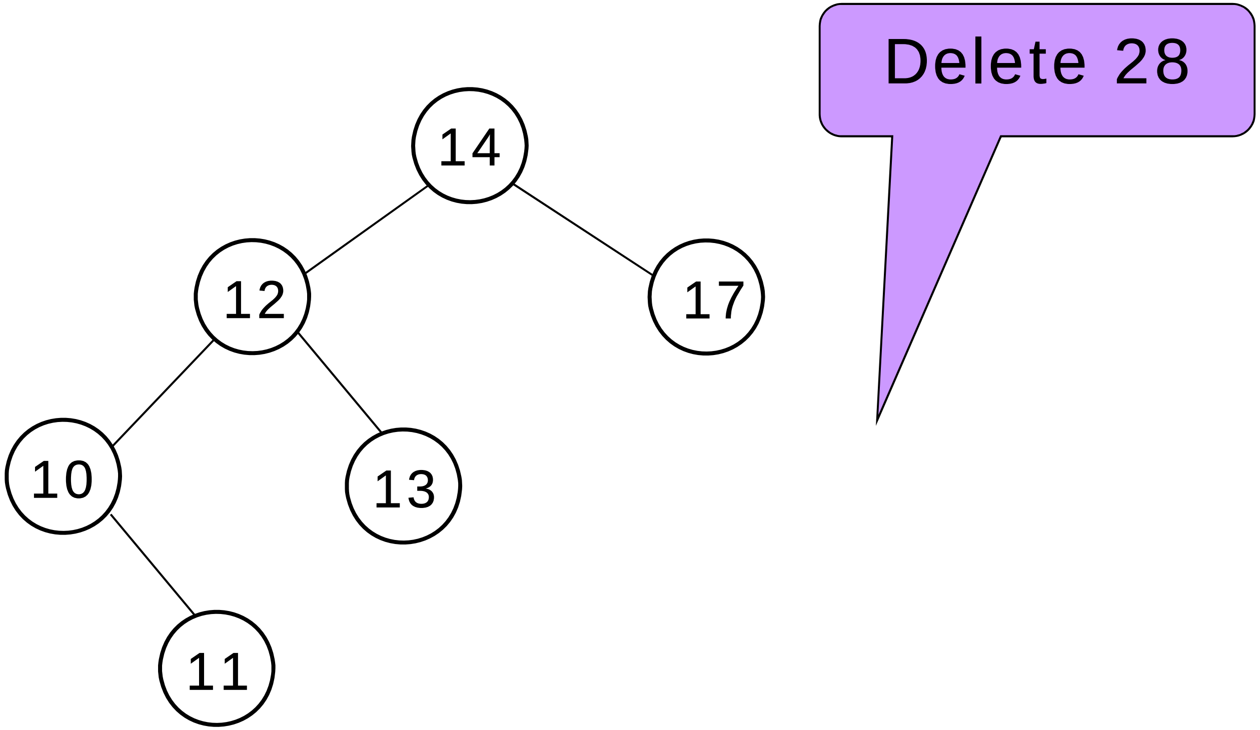
7.5.4 Delete

- 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

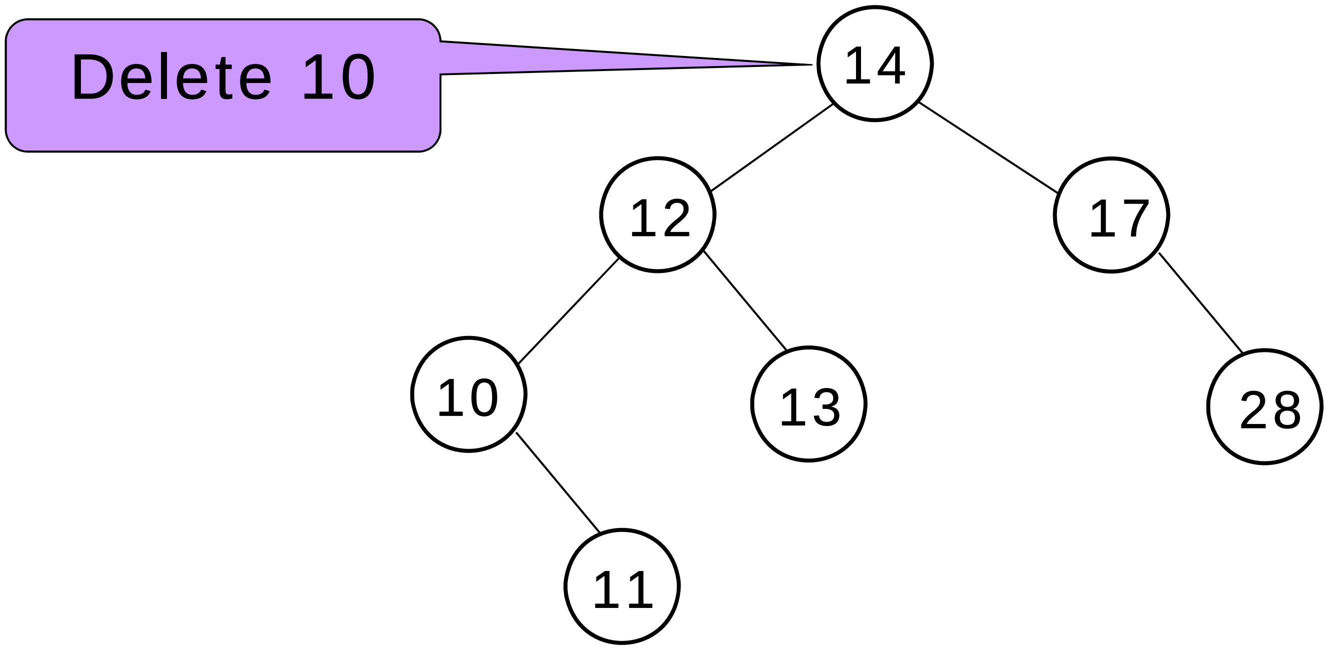
  (2) Make the child take place of the deleted node

7.5.4 Delete

- Deleting internal nodes of one child

  → (1) Delete the node

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

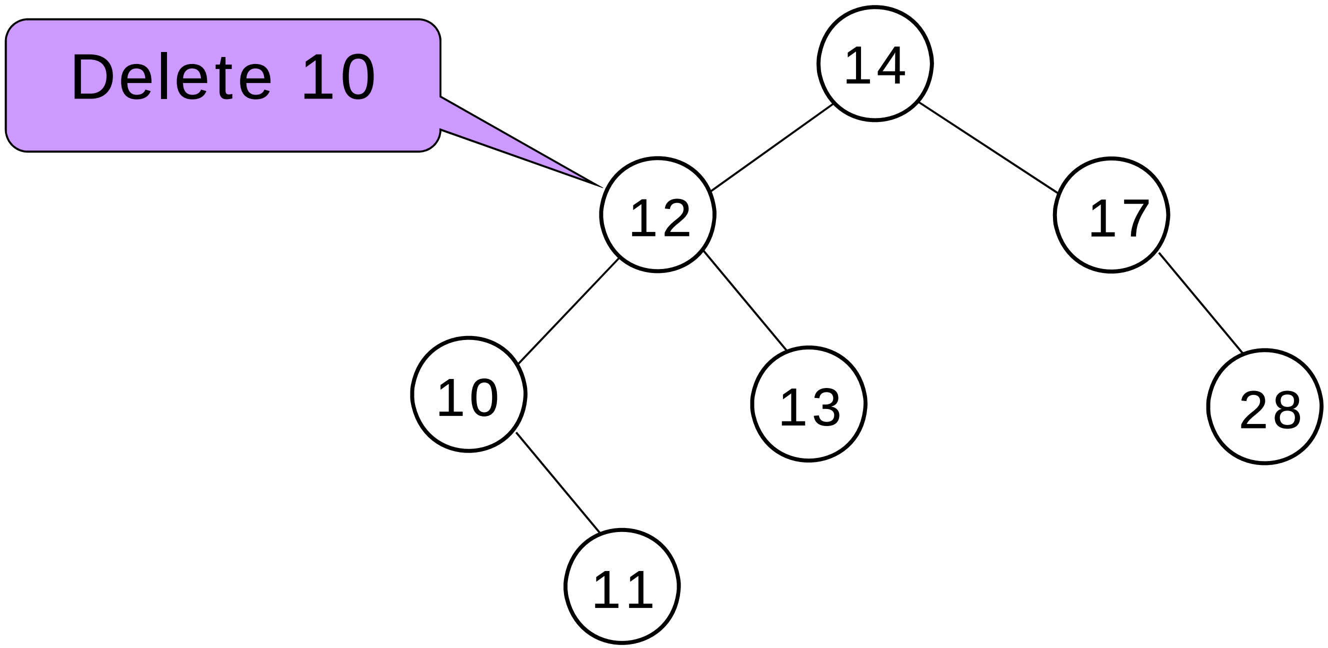


7.5.4 Delete

- Deleting internal nodes of one child

  → (1) Delete the node

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

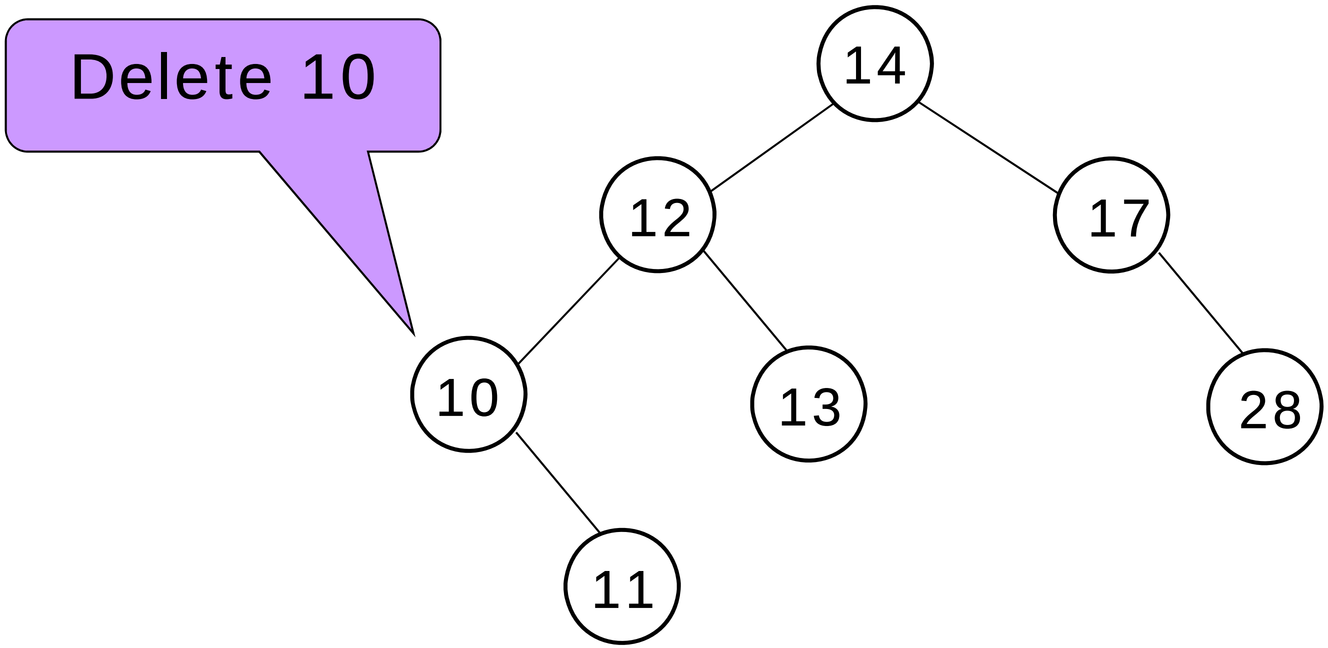


7.5.4 Delete

- Deleting internal nodes of one child

  → (1) Delete the node

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

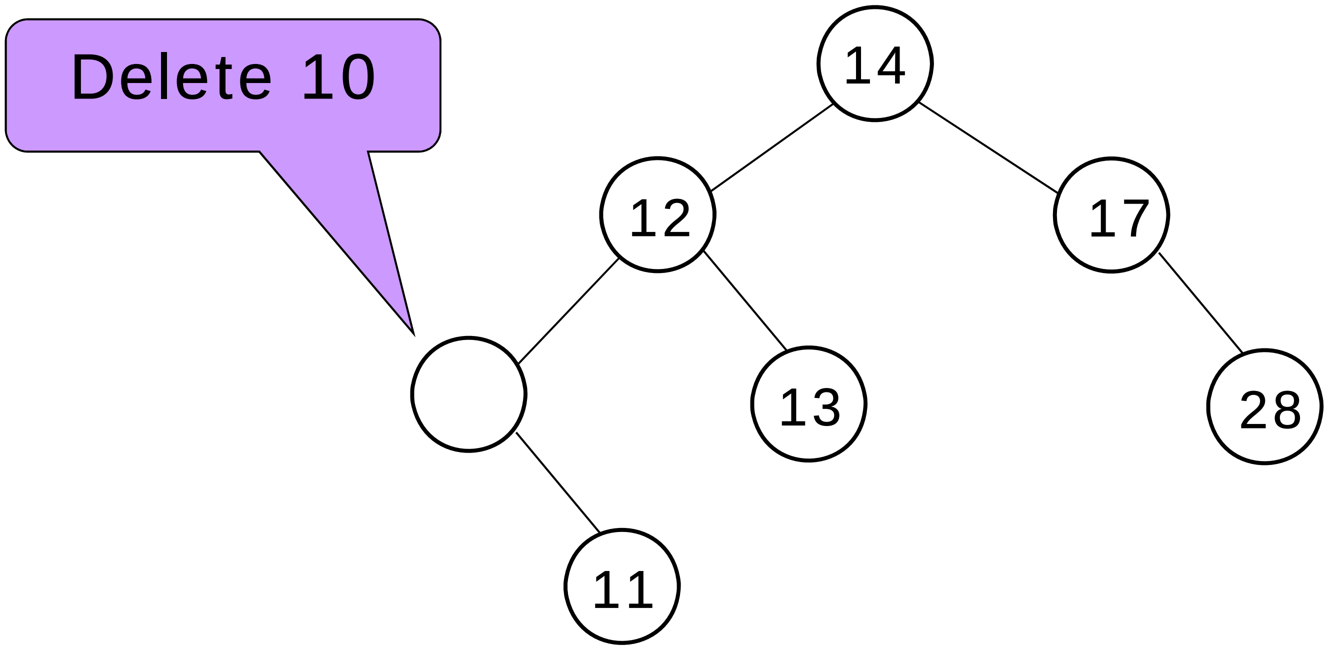


7.5.4 Delete

- Deleting internal nodes of one child

  → (1) Delete the node

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

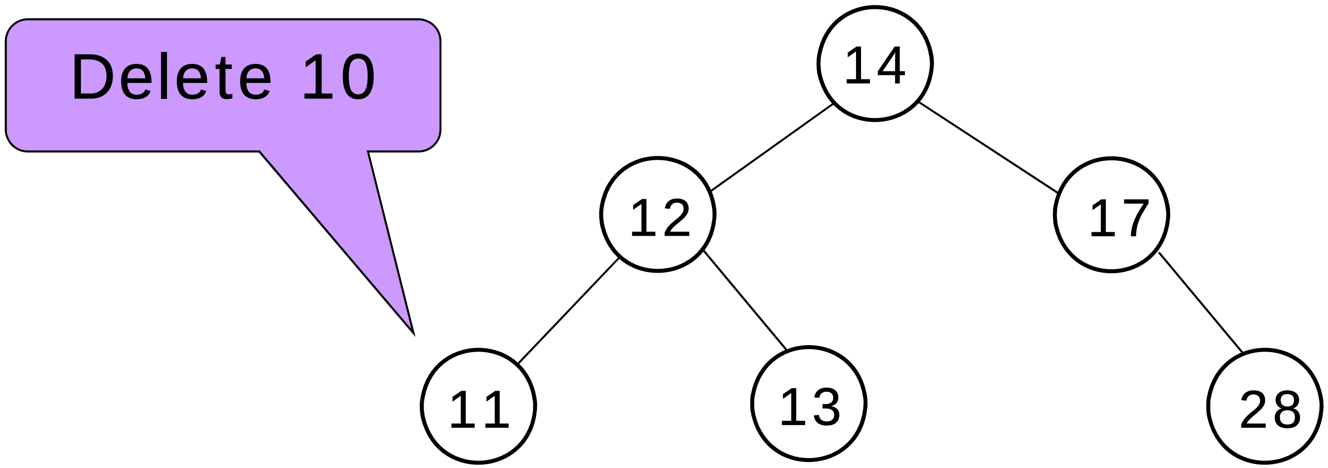


7.5.4 Delete

- 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

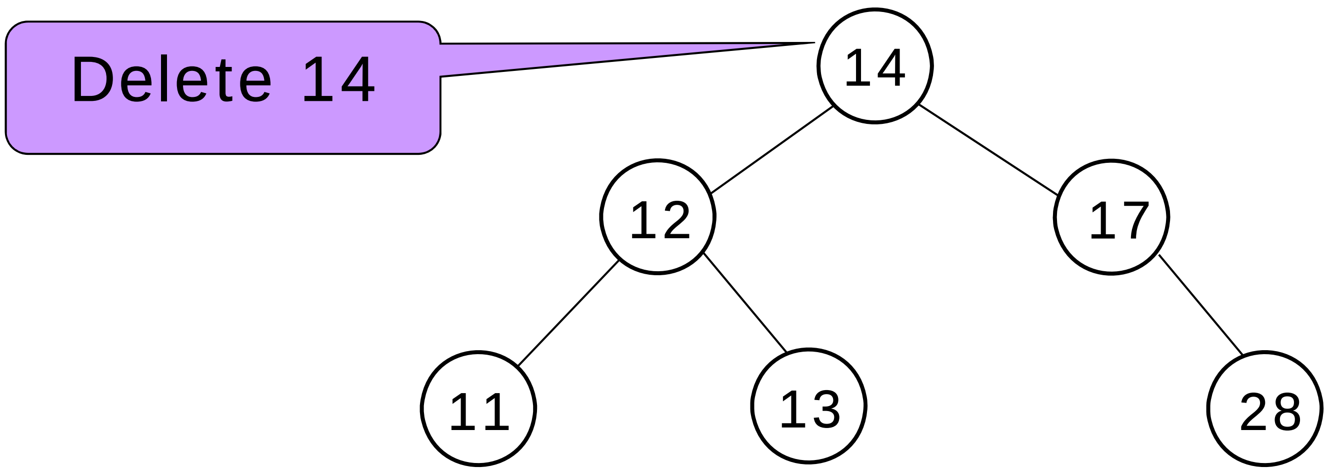
  (2) Move the maximum of its left subtree (or the minimum of its right subtree) to the node

7.5.4 Delete

- Deleting internal nodes with two childs

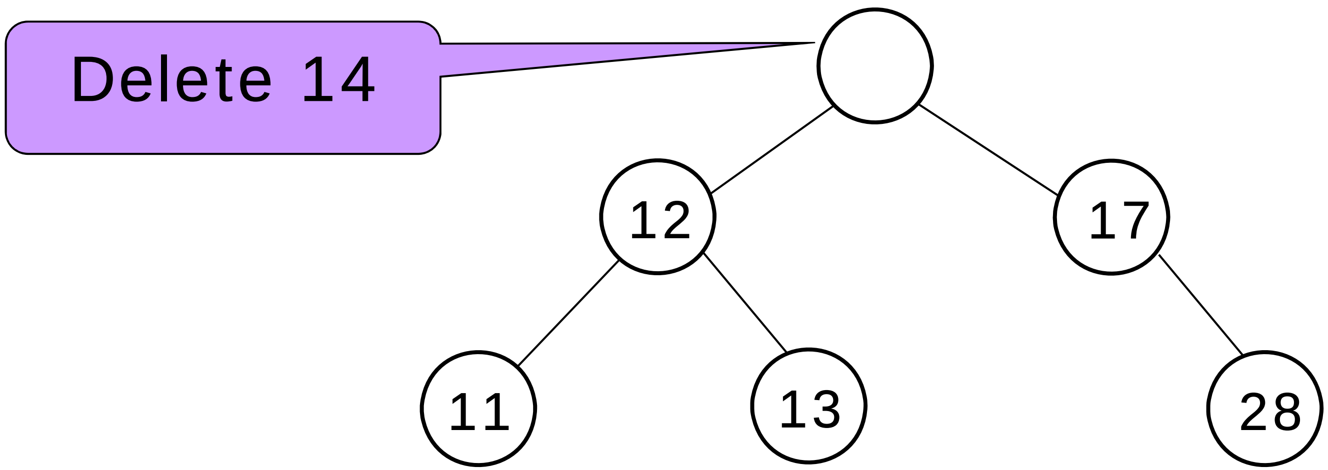
  → (1) Delete the node

  (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

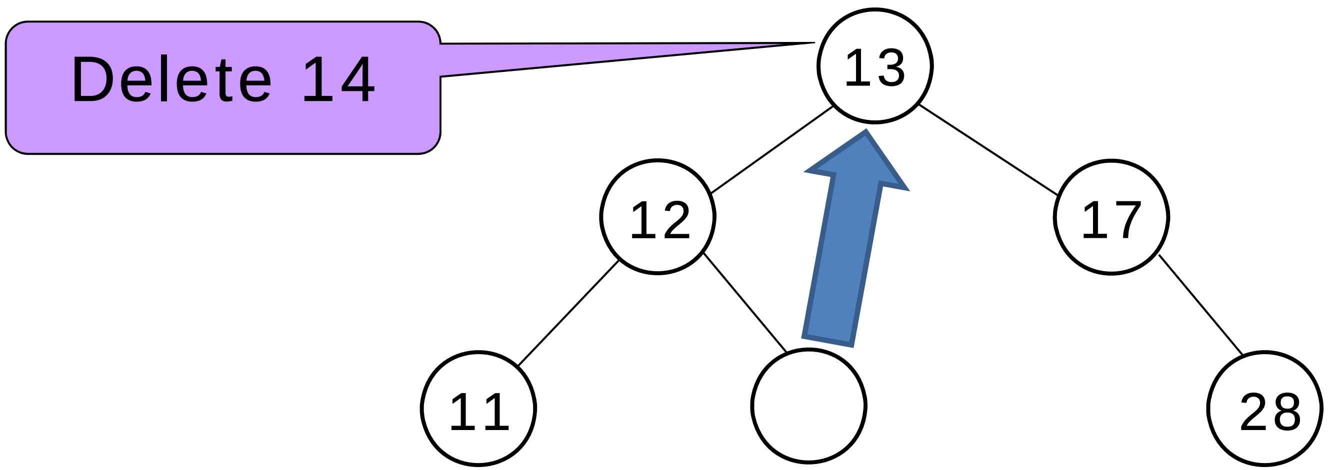


7.5.4 Delete

- 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

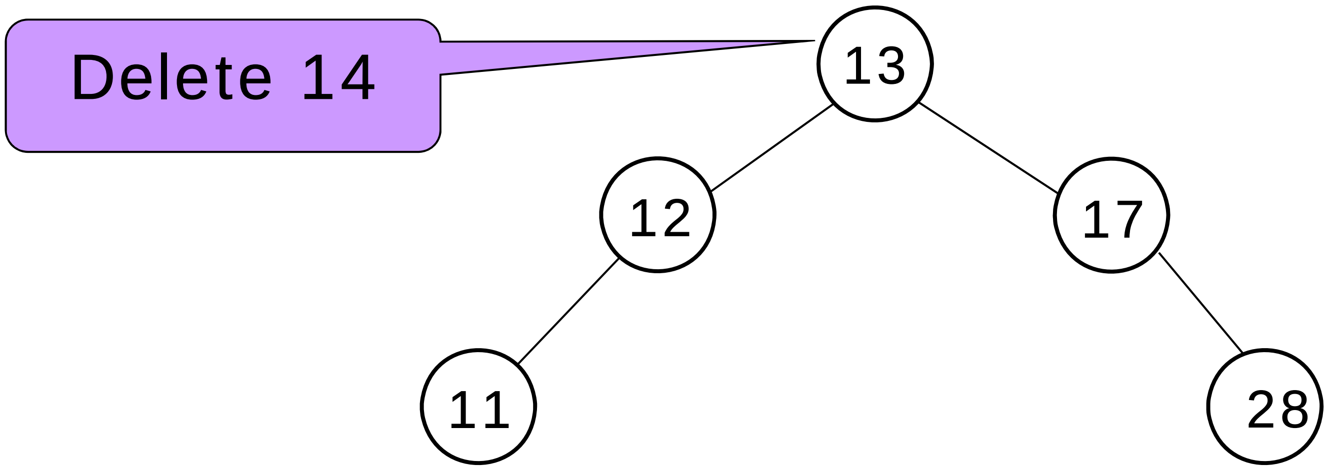


7.5.4 Delete

- 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

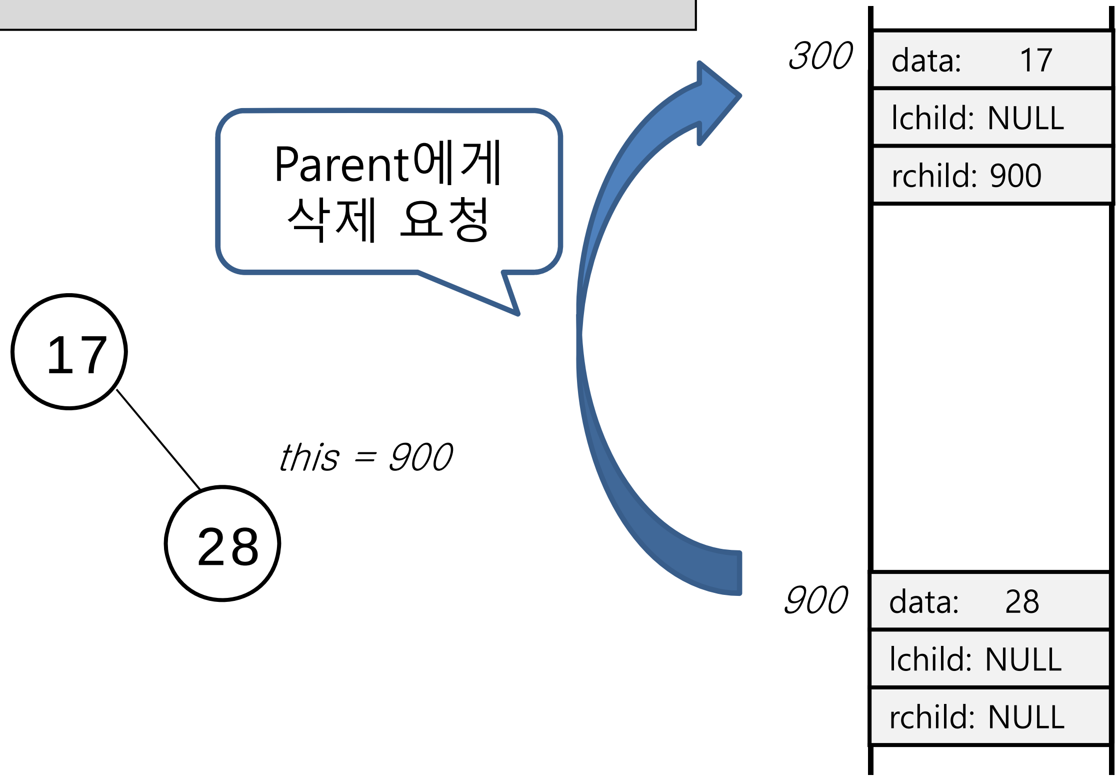
- Recursive implementation

```
int node::remove(int ndata)
{
//   key와 같으면
     if (this->key == ndata) {
          printf("Removing %d\n", ndata);
          // (1) 두 child가 다 NULL
          // (2) lchild만 NULL
          // (3) rchild만 NULL
          // (4) 둘 다 NULL이 아닌 경우
     }
//   key보다 크면
     else if (this->key < ndata) {
          this->rchild->remove(ndata);
     }
//   key보다 작으면
     else {
          this->lchild->remove(ndata);
     }
}
```

7.5.4 Delete

- Recursive implementation

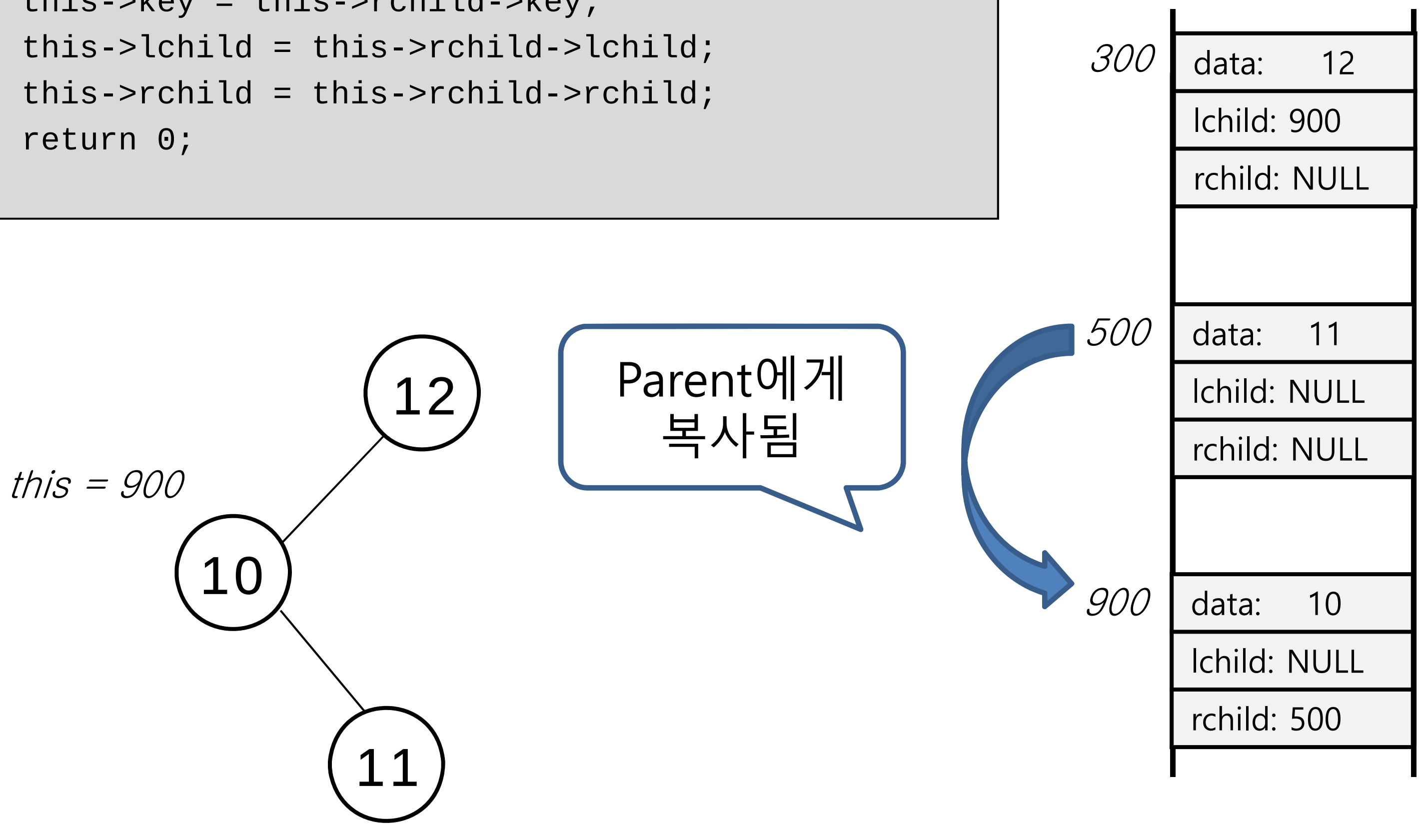
```
    //  (1) 두 child가 다 NULL → leaf node이면
        if (this->lchild == NULL && this->rchild == NULL)
            return 1;
```



7.5.4 Delete

- Recursive implementation

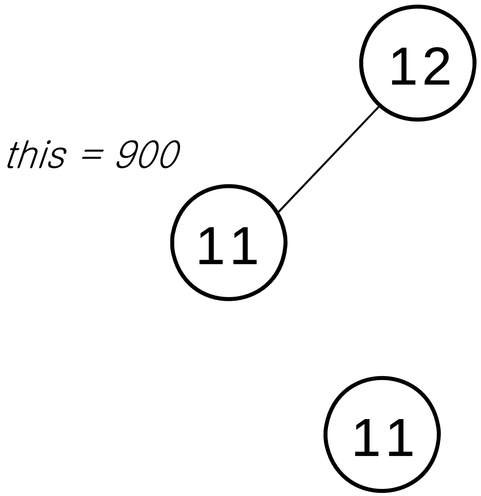
```
    //  (2) left child만 NULL이면
        if (this->lchild == NULL && this->rchild != NULL) {
            this->key = this->rchild->key;
            this->lchild = this->rchild->lchild;
            this->rchild = this->rchild->rchild;
            return 0;
        }
```

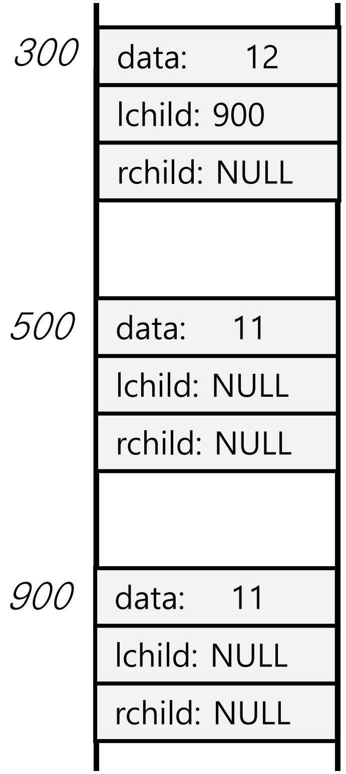


7.5.4 Delete

- Recursive implementation

```
    //  (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;
        }
```





7.5.4 Delete

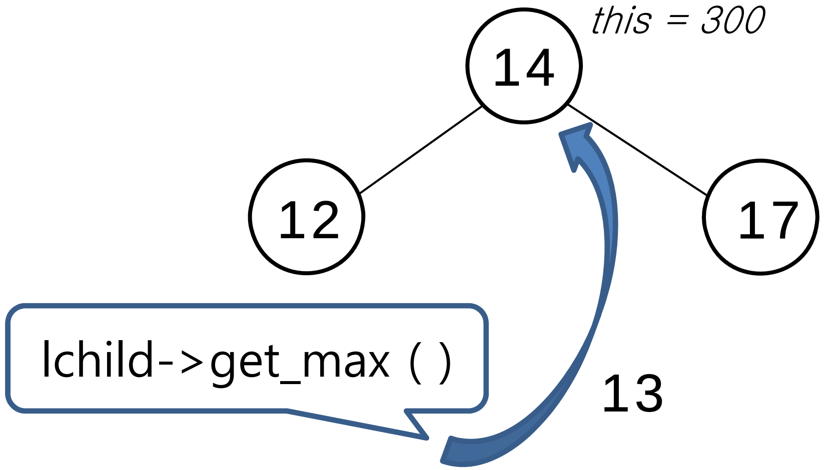
- Recursive implementation

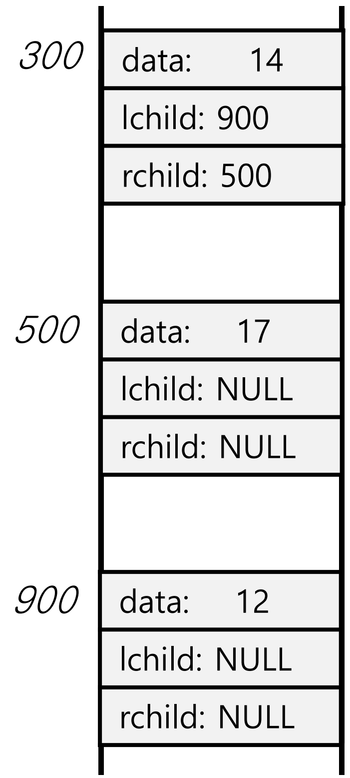
```
    //  (3) right child만 NULL이면
        if (this->lchild != NULL && this->rchild == NULL) {
            this->key = this->lchild->key;
            this->rchild = this->lchild->rchild;
            this->lchild = this->lchild->lchild;
            return 0;
        }
```

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;
        }
    }
```

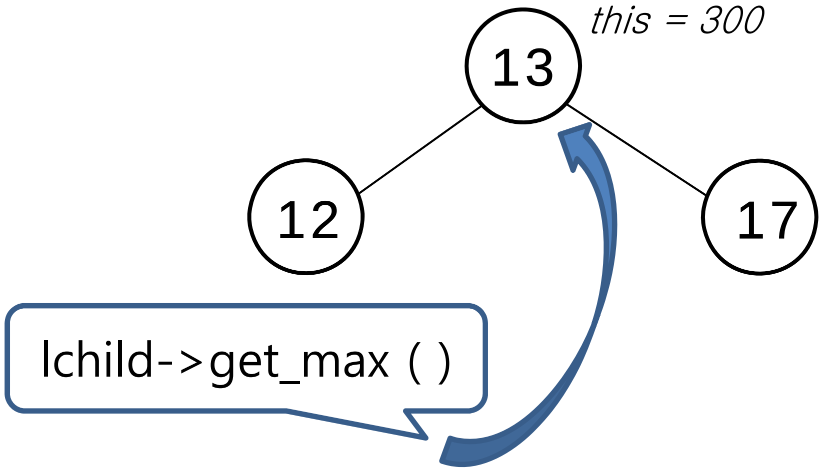


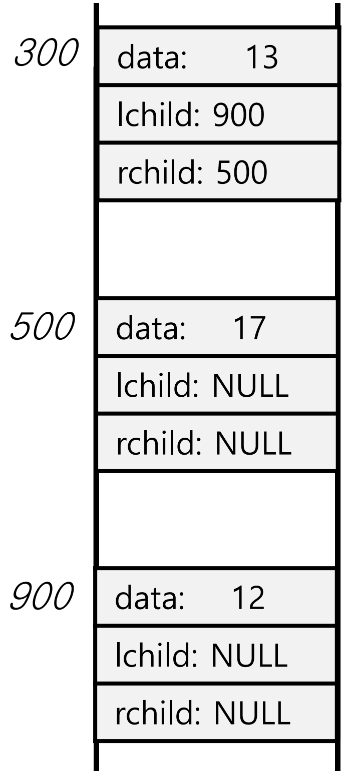


7.5.4 Delete

- Recursive implementation

```
//  (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;
    }
}
```

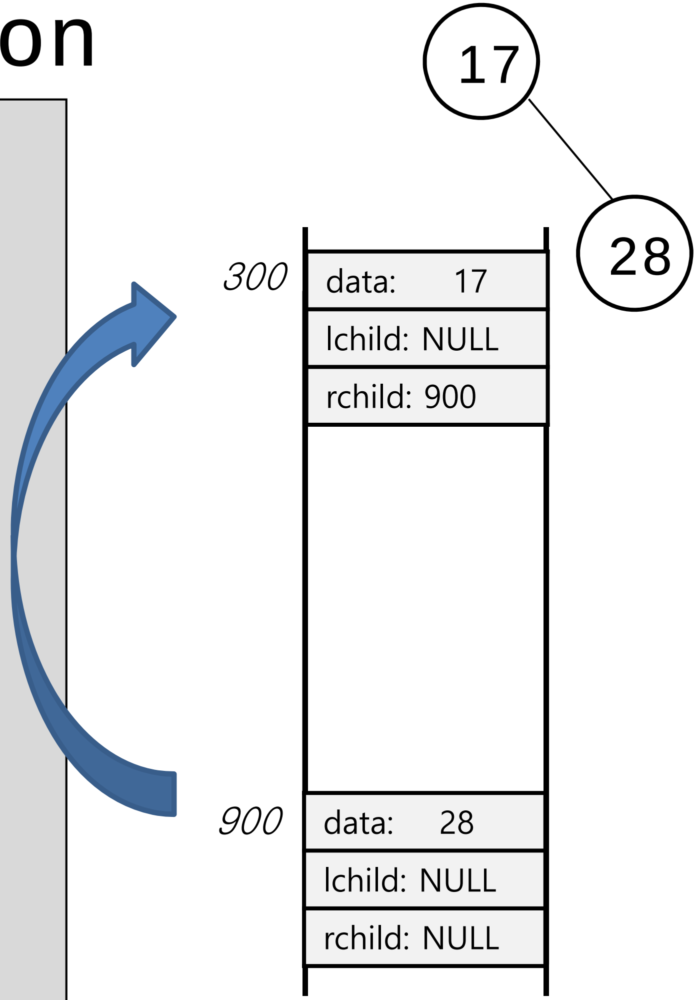




7.5.4 Delete

- Recursive implementation

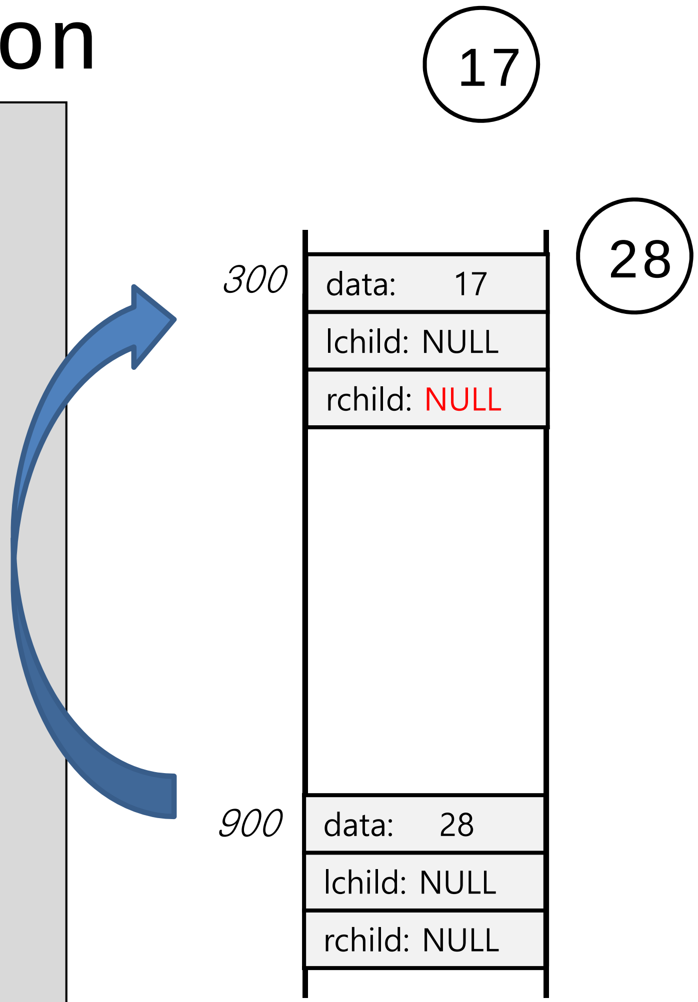
```
else if (this->key < ndata) {
    if (this->rchild != NULL) {
        if (this->rchild->remove(ndata))
            this->rchild = NULL;
    }
    else {
        printf("Not found %d in removing\n", ndata);
    }
    return 0;
}
else {
    if (this->lchild != NULL) {
        if (this->lchild->remove(ndata))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



7.5.4 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;
    }
```

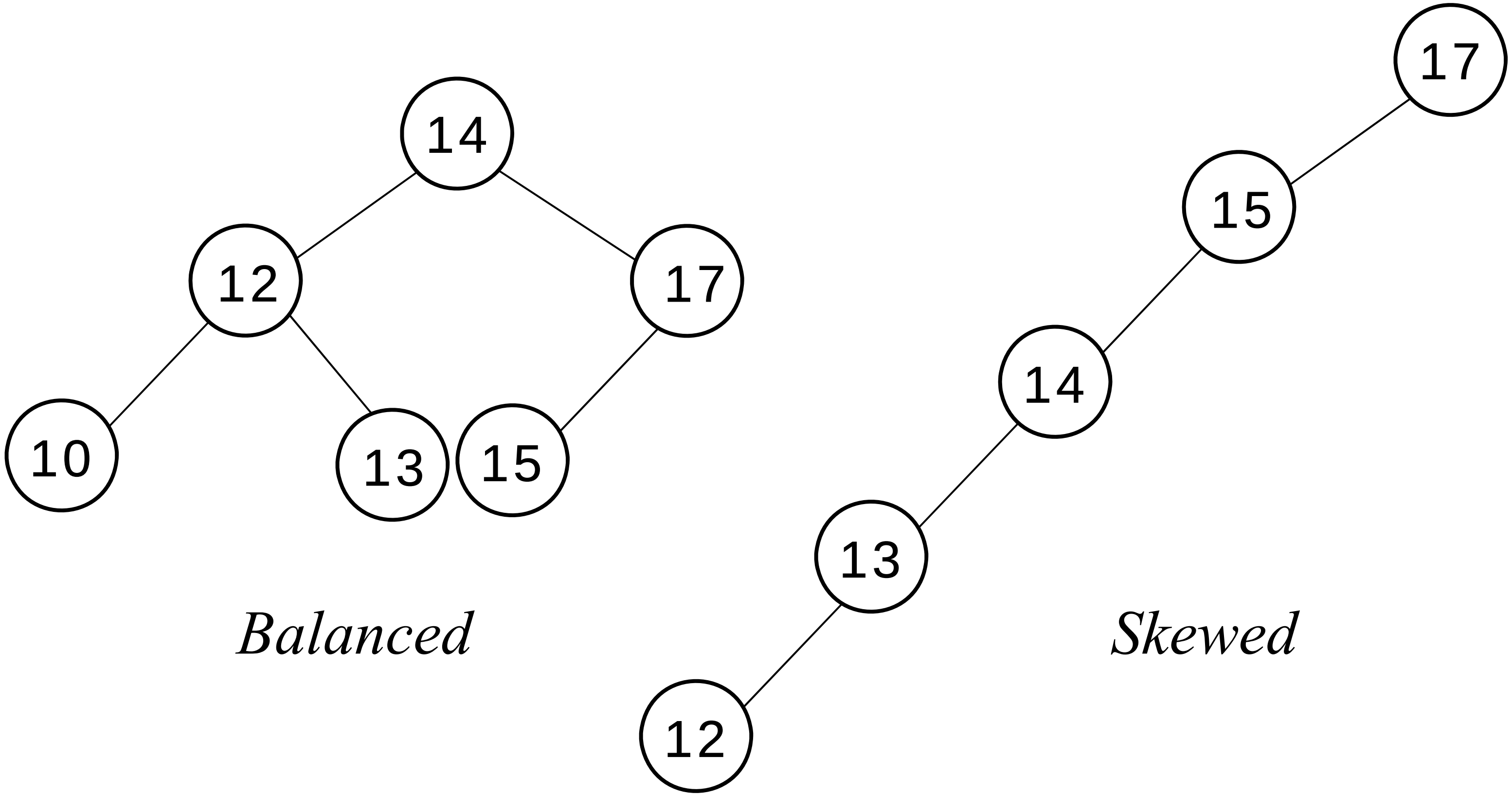


7.5.4 Delete

- Time complexity
  - Best case
    - A binary tree with n nodes has depth of log n.
    - At worst case, delete ends at the leaf nodes.
    - So, the best case time complexity is O(log n).
  - Worst case
    - A binary tree with n nodes has depth of n.
    - At worst case, delete ends at the leaf nodes.
    - So, the best case time complexity is O(n).

7.5.5 Time complexity

- Balanced (best) VS Skewed (worst)



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?

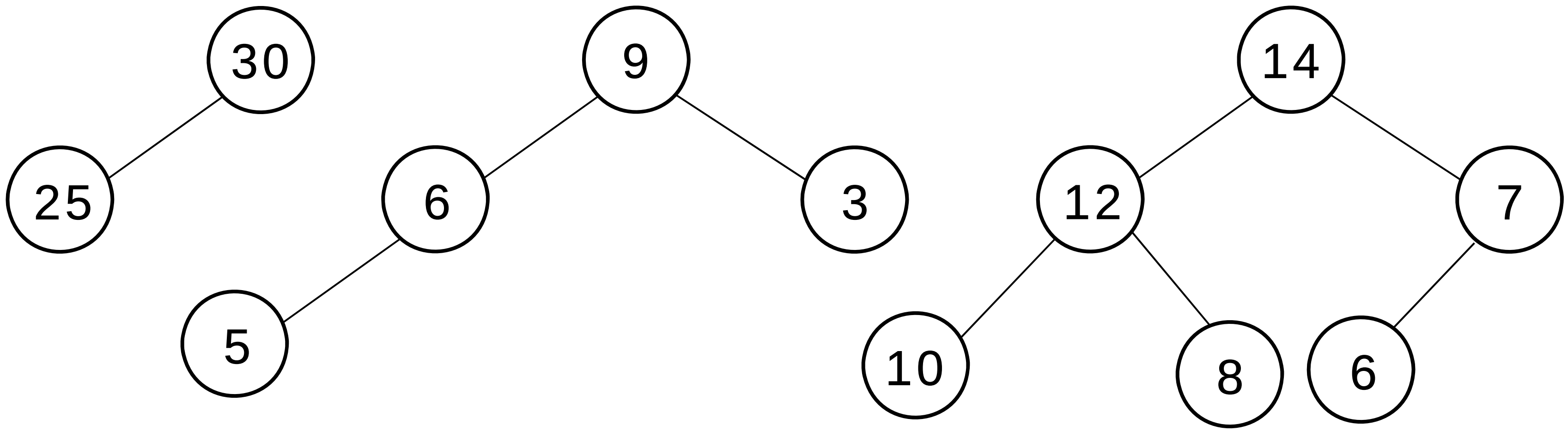
  → use tree!! (heap)

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

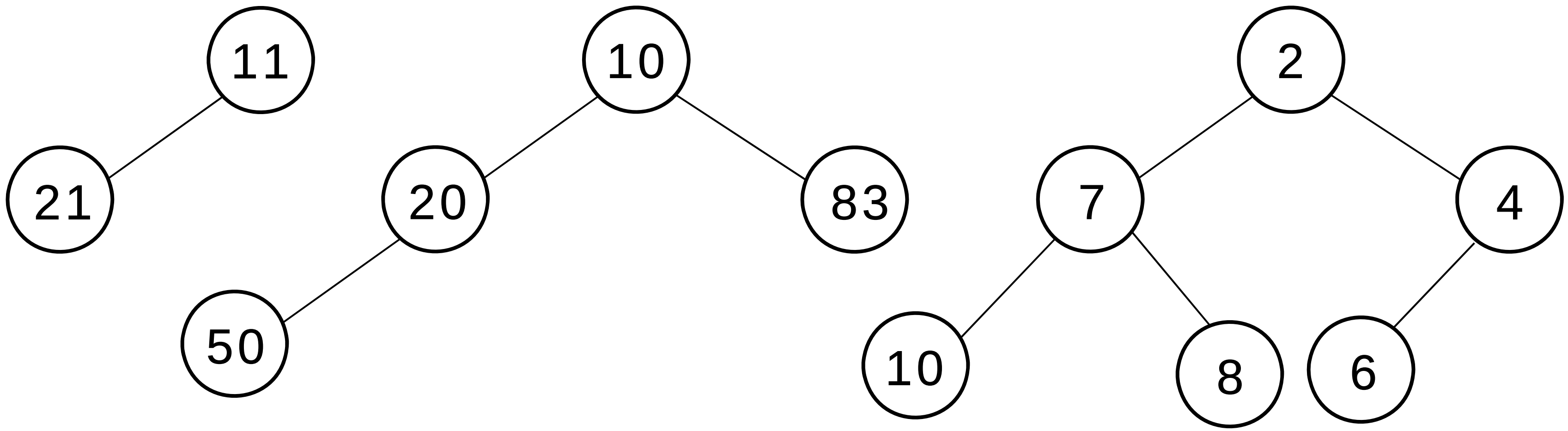
7.6.2 Definition of a Heap

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



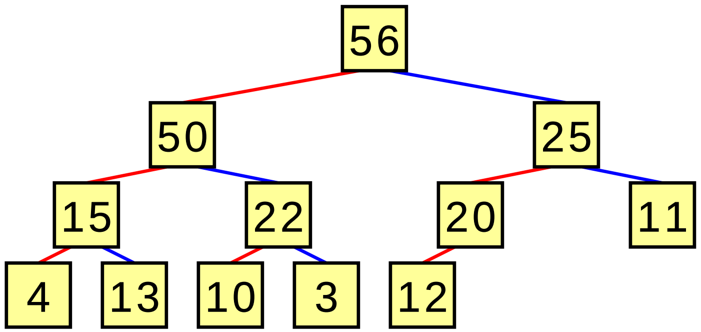
7.6.2 Definition of a Heap

- 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

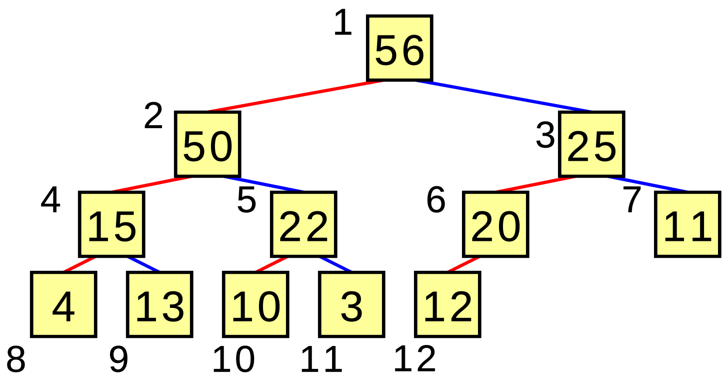
- Implementation of a heap
  - Implementation of a complete binary tree
    - Pointer-based
    - Array-based

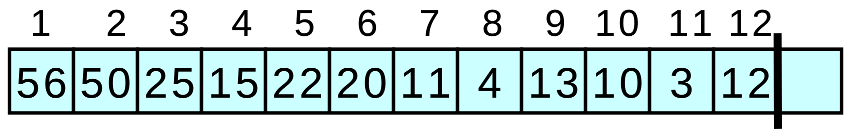


56	50	25	15	22	20	11	4	13	10	3	12	

7.6.2 Definition of a Heap

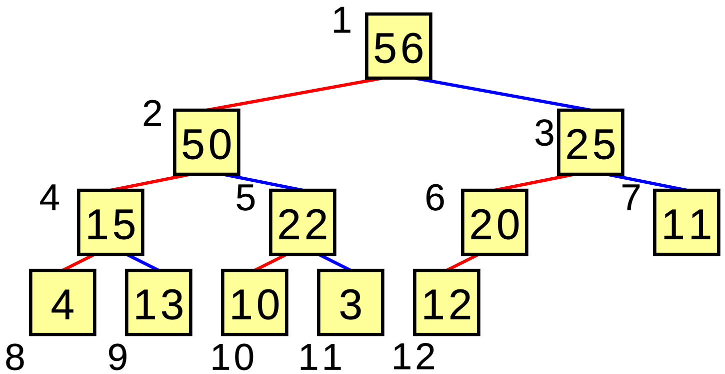
- Implementation of a heap
  - Index the nodes of a heap from top to down, from left to right
  - Index the elements of an array from 1

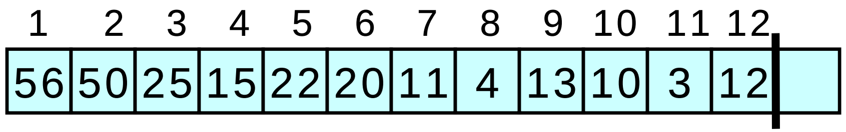




7.6.2 Definition of a Heap

- 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





7.6.2 Definition of a Heap

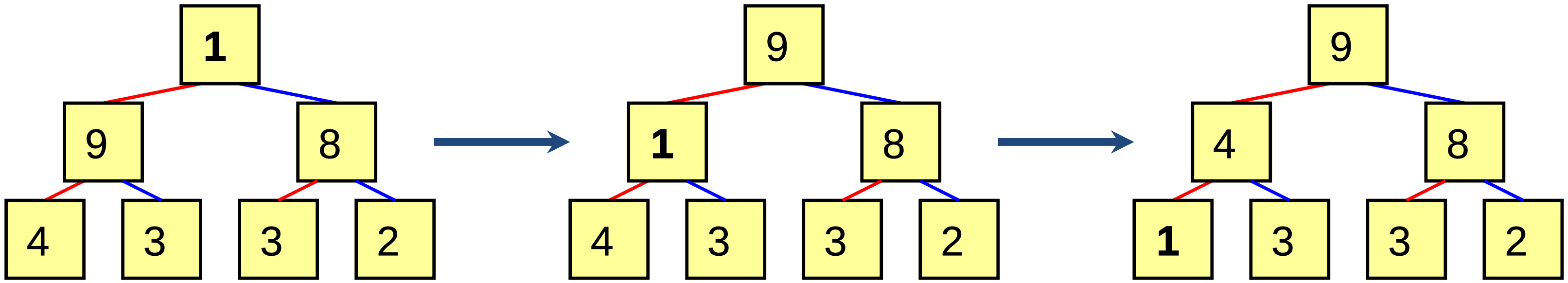
- Implementation of a heap

```
int *heap;
int n;
int cnt;
```

```
cnt = 0;
n = 1000;
heap = (int *)calloc(n, sizeof(int));
```

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 ( )

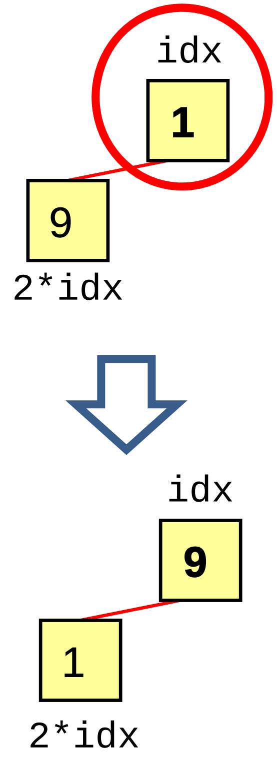


7.6.2 Definition of a Heap

- Topdown heapify (k) → max heap

```
void heapify_topdown(int idx)
{
    //    leaf node에 도착하면 끝
    if (2 * idx >= cnt)       return;

    if (2 * idx == cnt) {
        if (heap[idx] < heap[2 * idx])
            swap(&heap[idx], &heap[2 * idx]);
        return;
    }
    if (heap[2*idx] > heap[2*idx+1] && heap[2*idx] > heap[idx]) {
        swap(&heap[idx], &heap[2 * idx]);
        heapify_topdown(2 * idx);
    }
    else if (heap[2*idx+1] > heap[2*idx] && heap[2*idx+1] > heap[idx]) {
        swap(&heap[idx], &heap[2 * idx + 1]);
        heapify_topdown(2 * idx + 1);
    }
}
```

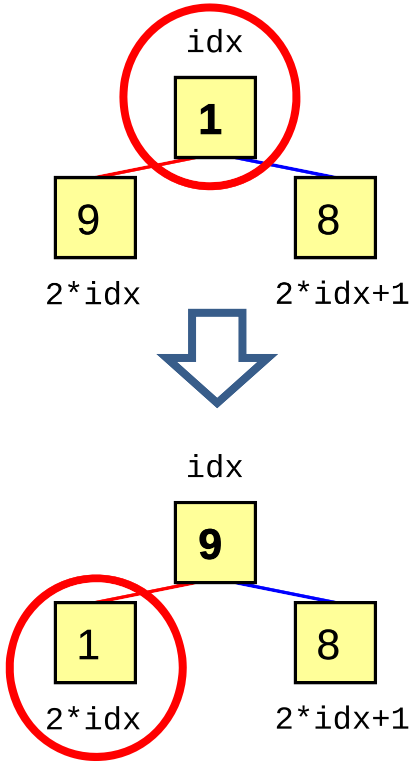


7.6.2 Definition of a Heap

- Topdown heapify (k) → max heap

```
void heapify_topdown(int idx)
{
    //    leaf node에 도착하면 끝
    if (2 * idx >= cnt)       return;

    if (2 * idx == cnt) {
        if (heap[idx] < heap[2 * idx])
            swap(&heap[idx], &heap[2 * idx]);
        return;
    }
    if (heap[2*idx] > heap[2*idx+1] && heap[2*idx] > heap[idx]) {
        swap(&heap[idx], &heap[2 * idx]);
        heapify_topdown(2 * idx);
    }
    else if (heap[2*idx+1] > heap[2*idx] && heap[2*idx+1] > heap[idx]) {
        swap(&heap[idx], &heap[2 * idx + 1]);
        heapify_topdown(2 * idx + 1);
    }
}
```

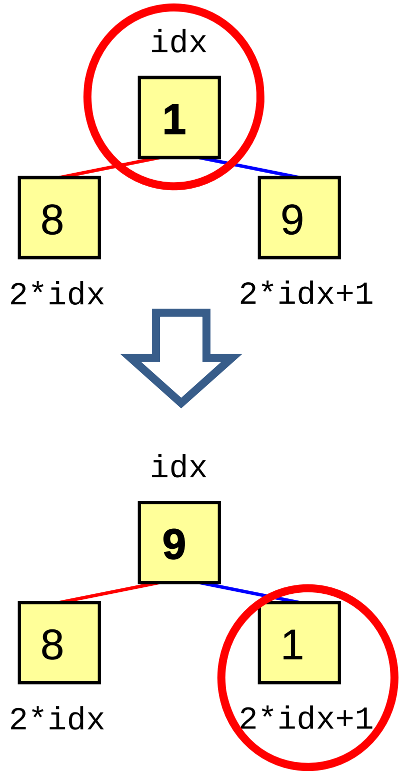


7.6.2 Definition of a Heap

- Topdown heapify (k) → max heap

```
void heapify_topdown(int idx)  //  recursive
{
    //    leaf node에 도착하면 끝
    if (2 * idx >= cnt)       return;

    if (2 * idx == cnt) {
        if (heap[idx] < heap[2 * idx])
            swap(&heap[idx], &heap[2 * idx]);
        return;
    }
    if (heap[2*idx] > heap[2*idx+1] && heap[2*idx] > heap[idx]) {
        swap(&heap[idx], &heap[2 * idx]);
        heapify_topdown(2 * idx);
    }
    else if (heap[2*idx+1] > heap[2*idx] && heap[2*idx+1] > heap[idx]) {
        swap(&heap[idx], &heap[2 * idx + 1]);
        heapify_topdown(2 * idx + 1);
    }
}
```



7.6.2 Definition of a Heap

- Topdown heapify (k) → max heap

```
void heapify_topdown(int idx) //  iterative
{
    while (2 * idx < cnt) {
        if (2 * idx == cnt) { //        lchild만 있는 경우
            if (heap[idx] < heap[2 * idx])
                swap(&(heap[idx]), &(heap[2 * idx]));
            break;
        }
        if (heap[2 * idx] > heap[2 * idx + 1] && heap[idx] < heap[2 * idx])  {
            swap(&(heap[idx]), &(heap[2 * idx]));
            idx = 2 * idx;
        }
        else if (heap[2 * idx] < heap[2 * idx + 1] && heap[idx] < heap[2 * idx + 1]) {
            swap(&(heap[idx]), &(heap[2 * idx + 1]));
            idx = 2 * idx + 1;
        }
        else
            break;
    }
}
```

7.6.2 Definition of a Heap

- Bottomup heapify (k) → max heap

```
void heapify_bottomup(int idx)  //  recursive
{
    //    root node에 도착하면 끝
    if (idx == 1)
        return;

    if (heap[idx / 2] < heap[idx]) {
        swap(&heap[idx / 2], &heap[idx]);
        heapify_bottomup(idx / 2);
    }
}
```

7.6.2 Definition of a Heap

- Bottomup heapify (k) → max heap

```
void heapify_bottomup(int idx)  //  iterative
{
    //    root node에 도착하면 끝
    while (idx > 1) {
        if (heap[idx / 2] < heap[idx]) {
            swap(&heap[idx / 2], &heap[idx]);
            idx = (idx / 2);
        }
        else
            break;
    }
}
```

7.6.3 Insertion into a Heap

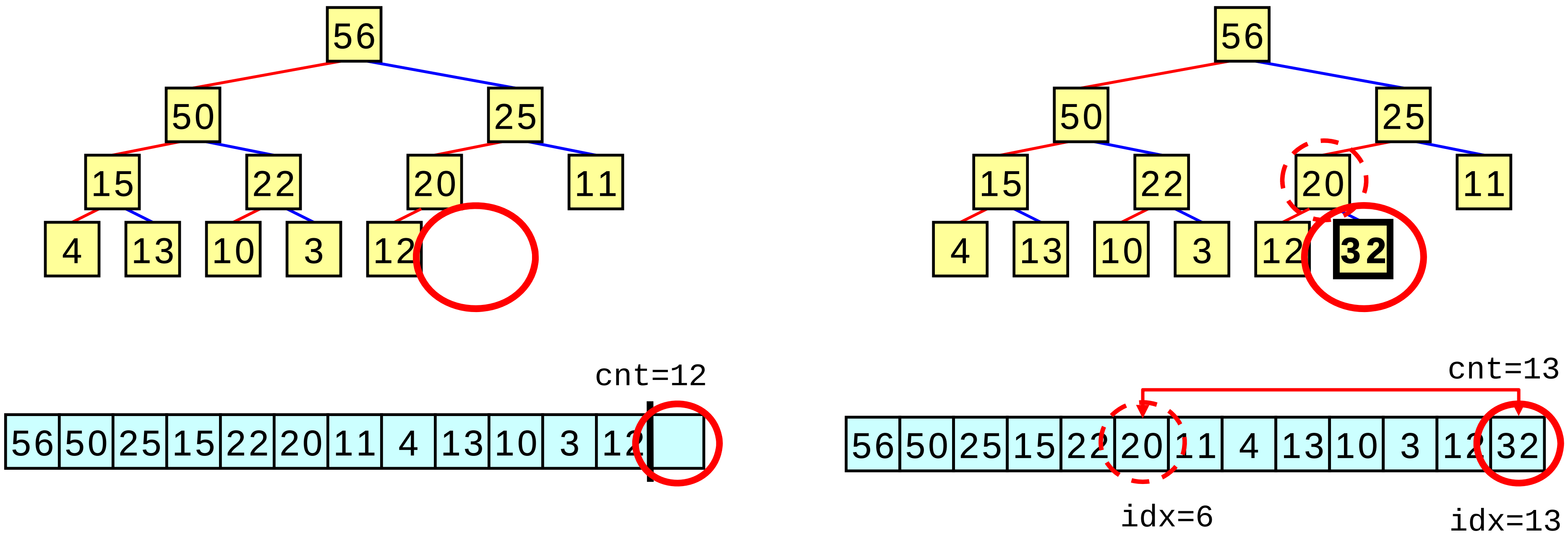
- Insert an element to a max heap

  (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

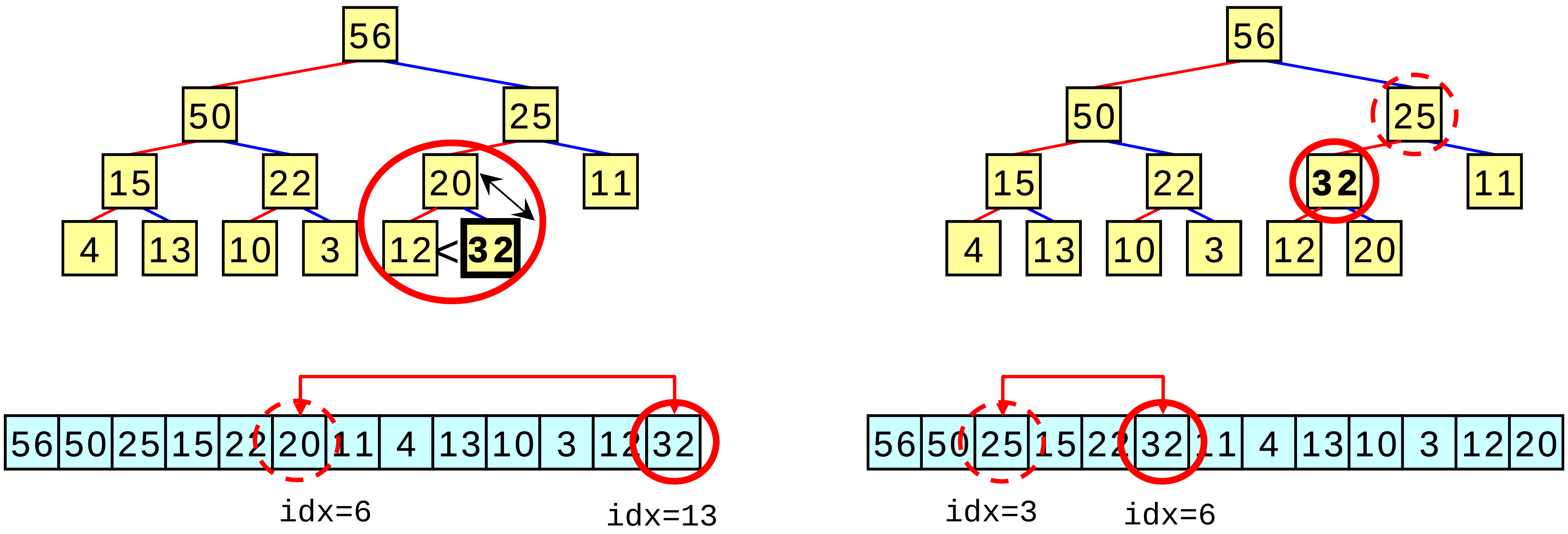
- Insert an element to a max heap
  (1) Insert an element to the last position of the heap (no longer heap)



7.6.3 Insertion into a Heap

- Insert an element to a max heap

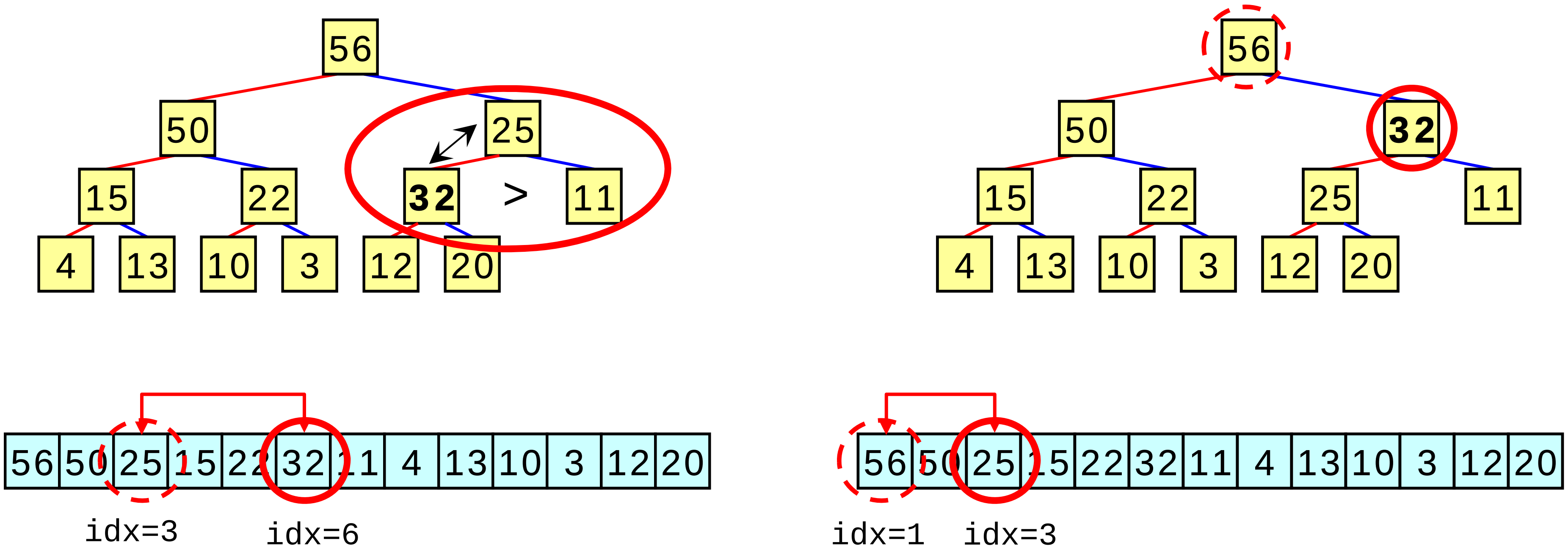
  (2) Using heapify ( ), reorganize the newly inserted heap to a heap



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

- Insert an element to a max heap

```
void insert(int ndata)
{
    cnt++;
    heap[cnt] = ndata;

    heapify_bottomup(cnt);
}
```

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

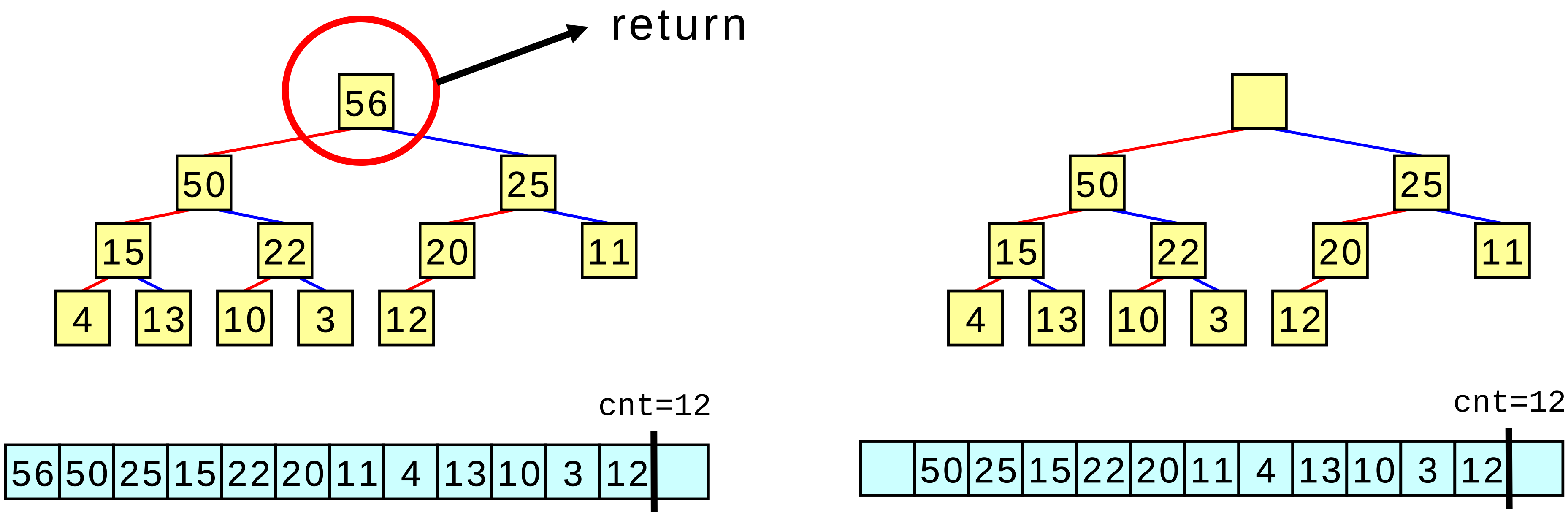
  (2) Move the element of the last node to the root node and remove the last node

  (3) Apply Heapify ( ) to maintain the heap

7.6.4 Deletion from a Heap

- Delete from a max heap

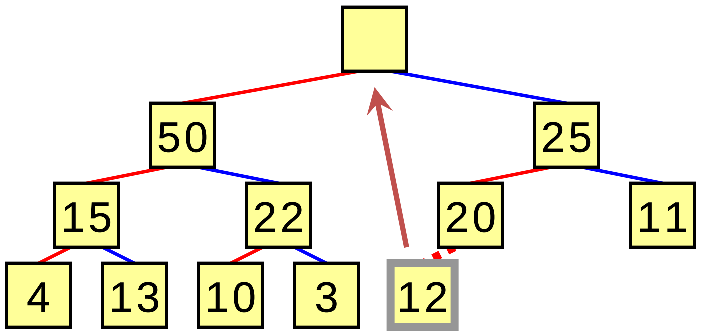
  (1) Remove the root of heap and return the element of the root node

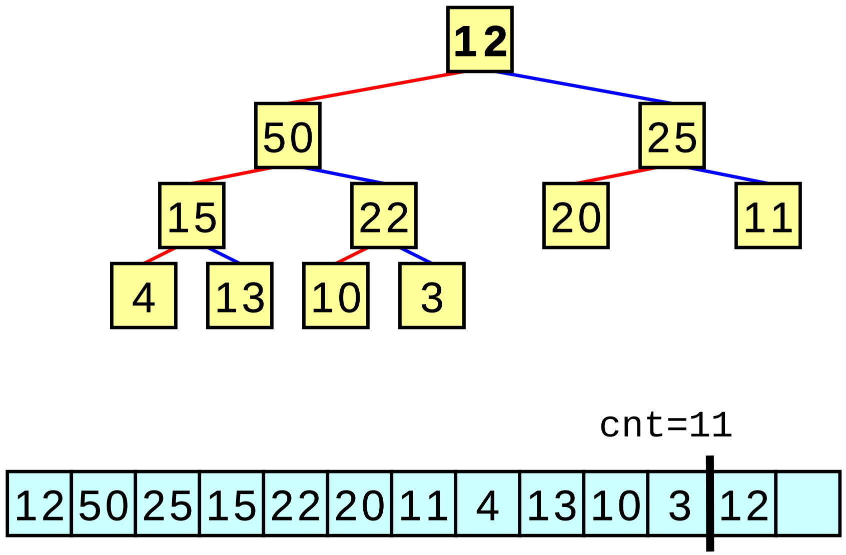


7.6.4 Deletion from a Heap

- Delete from a max heap

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

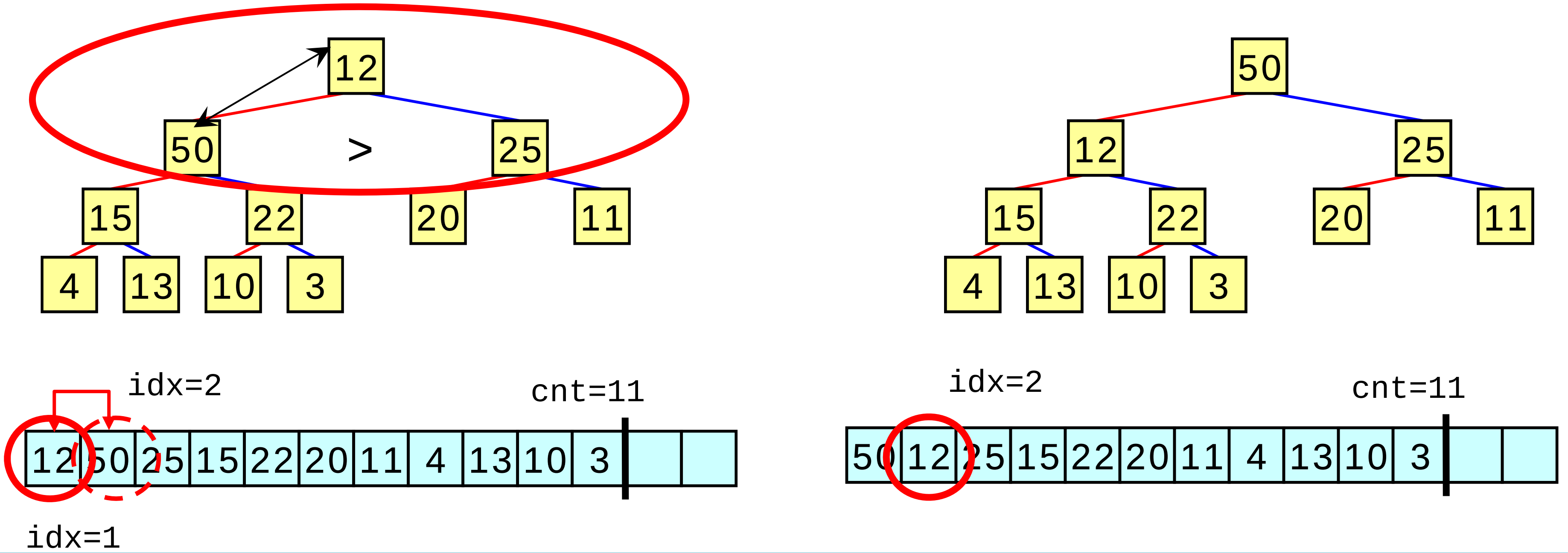




7.6.4 Deletion from a Heap

- Delete from a max heap

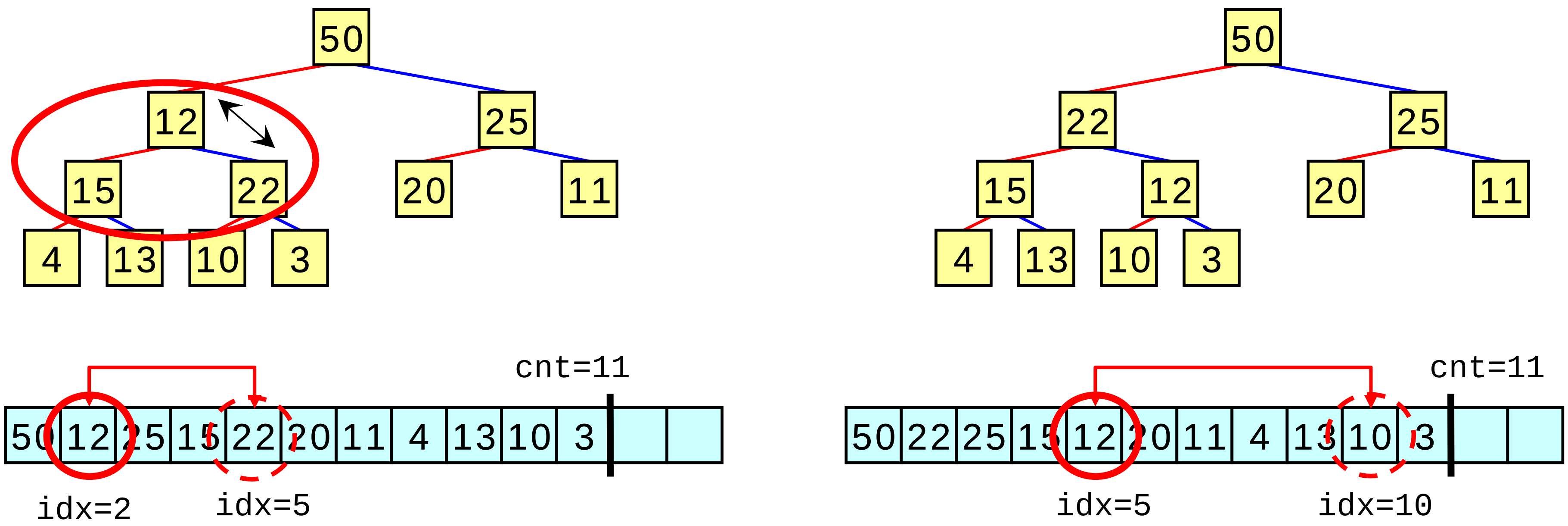
  (3) Apply heapify ( ) to maintain the structure of max heap



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

- Delete from a max heap

```
int remove ()
{
    int temp = heap[1];
    heap[1] = heap[cnt - 1];
    cnt--;

    heapify_topdown(1);
    return temp;
}
```

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)**		

Contents

Contents