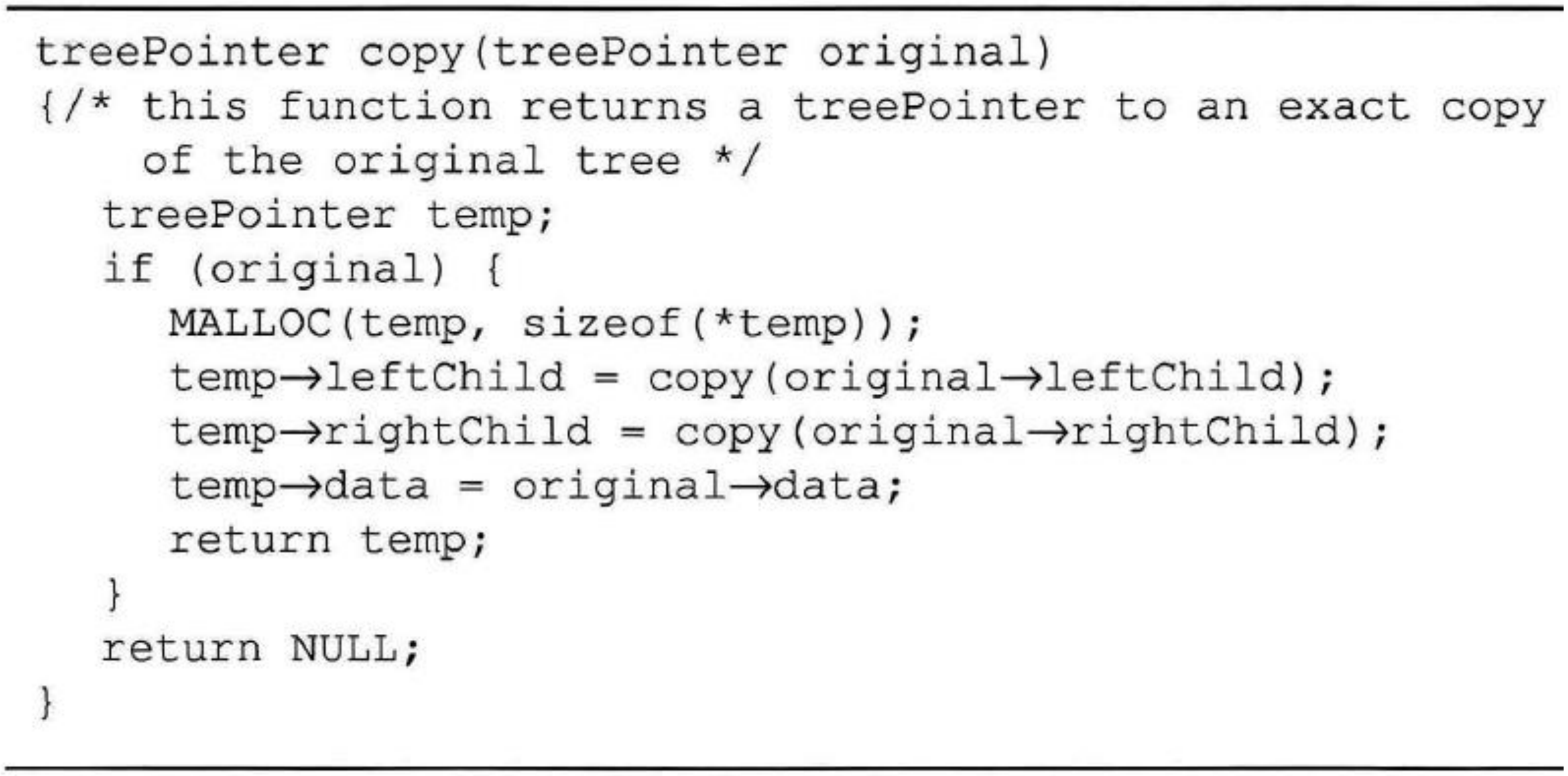
Chap 5. Trees (2)

Contents

5.4 Additional Binary Tree Operations

5.4.1 Copying Binary trees

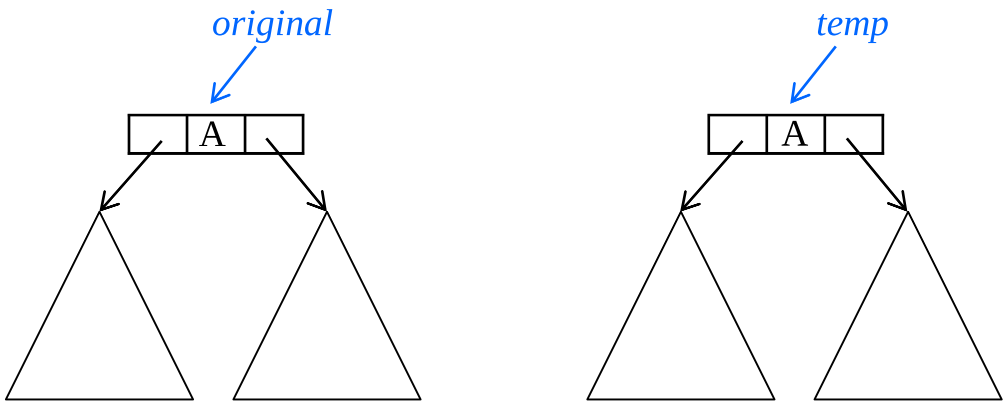
- A slightly modified version of *postorder* traversal



```
treePointer copy(treePointer original)
{/* this function returns a treePointer to an exact copy
    of the original tree */
   treePointer temp;
   if (original) {
      MALLOC(temp, sizeof(*temp));
      temp→leftChild = copy(original→leftChild);
      temp→rightChild = copy(original→rightChild);
      temp→data = original→data;
      return temp;
   }
   return NULL;
}
```

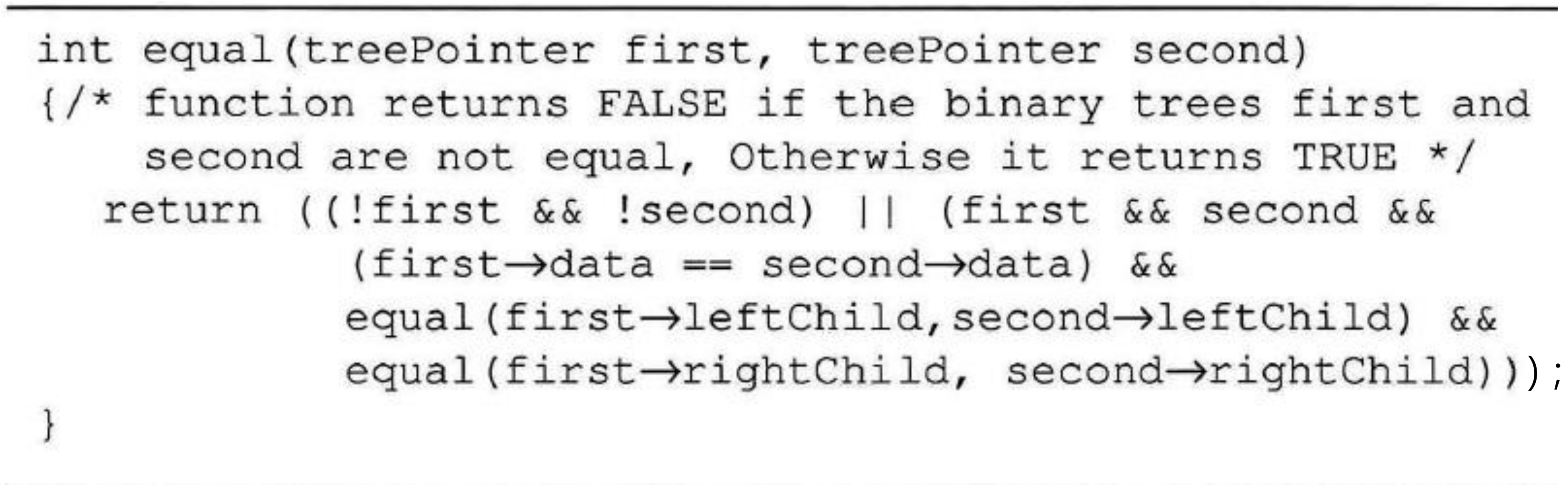
Program 5.6: Copying a binary tree

```
treePointer copy(treePointer original)
{
   treePointer temp;
   if (original) {
      MALLOC(temp, sizeof(*temp));
      temp→leftChild = copy(original→leftChild);    L
      temp→rightChild = copy(original→rightChild); R
      temp→data = original→data;                    V
      return temp;
   }
   return NULL;
}
```



5.4.2 Testing Equality

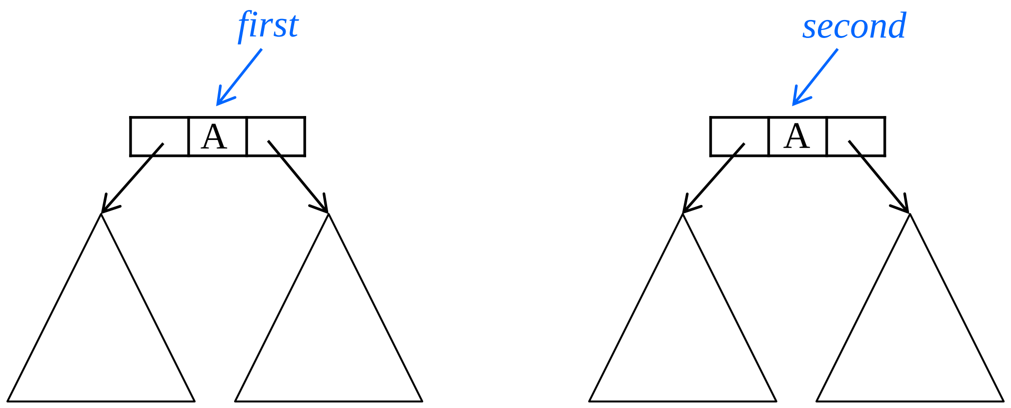
- Determining the equivalence of two binary trees
- Equivalent binary trees have the *same structure* and the *same information* in the corresponding nodes.
- A modification of *preorder* traversal



```
int equal(treePointer first, treePointer second)
{/* function returns FALSE if the binary trees first and
    second are not equal, Otherwise it returns TRUE */
  return ((!first && !second) || (first && second &&
          (first→data == second→data) &&
          equal(first→leftChild,second→leftChild) &&
          equal(first→rightChild, second→rightChild)));
}
```

Program 5.7: Testing for equality of binary trees

```
int equal(treePointer first, treePointer second)
{/* function returns FALSE if the binary trees first and
    second are not equal, Otherwise it returns TRUE */
  return ((!first && !second) || (first && second &&
             (first→data == second→data) &&                V
             equal(first→leftChild,second→leftChild) &&      L
             equal(first→rightChild, second→rightChild)));   R
}
```



5.4.3 The Satisfiability Problem

- Consider the set of formulas from $\{x_1, .., x_n\}$ and $\{\wedge$ (and), $\vee$ (or), $\neg$(not) $\}$

- The *variables* are *Boolean variables*
  - Have only two possible values, *true* or *false*

- Set of expressions are defined by the following rules
  - A variable is an expression
  - If x and y are expression, then $\neg x$, $x \wedge y$, $x \vee y$ are expressions
  - Parentheses can be used to alter the normal order of evaluation

- Propositional calculus: $x_1 \vee (x_2 \wedge \neg x_3)$
  - If x_1 and x_3 are *false* and x_2 is *true*, it is *true*

- ***The satisfiability problem***
  - Is there an assignment of values to the variables that causes the value of the expression to be true?

- The most obvious algorithm
  - let (x_1, .., x_n) take on all possible combinations of *true* and *false* values and to check the formula for each combination
    - O($g2^n$), or exponential time, where g is the time to substitute values for x_1, x_2, …, x_n and evaluate the expression.
  - ***Postorder*** evaluation

- $(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_3) \vee \neg x_3$

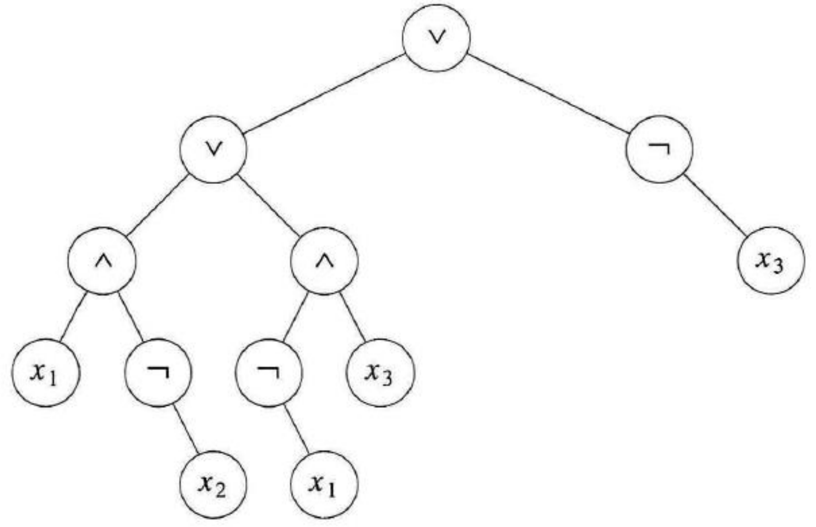


Figure 5.18: Propositional formula in a binary tree

For $n = 3$,
All possible combinations :
(*true*, *true*, *true*)
(*true*, *true*, *false*)
(*true*, *false*, *true*)
(*true*, *false*, *false*)
(*false*, *true*, *true*)
(*false*, *true*, *false*)
(*false*, *false*, *true*)
(*false*, *false*, *false*)

```
typedef enum {not,and,or,true,false} logical;
typedef struct node *treePointer;
typedef struct node {
        treePointer leftChild;
        logical     data;     // the value of a variable or an operator
        short int   value;    // TRUE/FALSE
        treePointer rightChild;
        } node;
```

leftChild	*data*	*value*	*rightChild*

```
for (all $2^n$ possible combinations) {
  generate the next combination;
  replace the variables by their values;
  evaluate root by traversing it in postorder;
  if (root→value) {
    printf(<combination>);
    return;
  }
}
printf("No satisfiable combination\n");
```

Program 5.8: First version of satisfiability algorithm

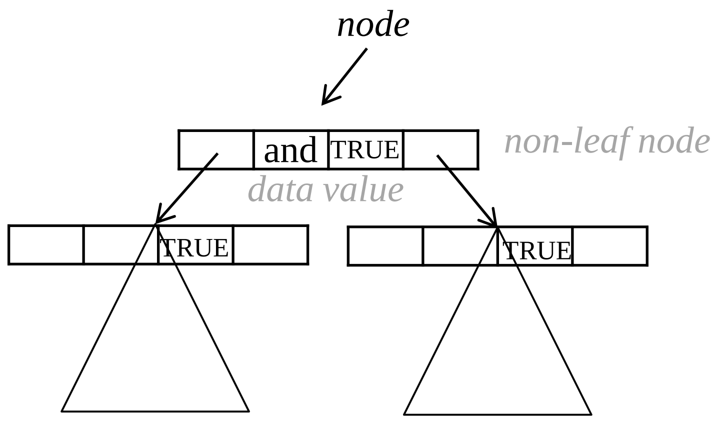
```
void postOrderEval(treePointer node)
{/* modified post order traversal to evaluate a
    propositional calculus tree */
   if (node) {
      postOrderEval(node→leftChild);
      postOrderEval(node→rightChild);
      switch(node→data) {
         case not:   node→value =
                     !node→rightChild→value;
                     break;
         case and:   node→value =
                     node→rightChild→value &&
                     node→leftChild→value;
                     break;
         case or:    node→value =
                     node→rightChild→value ||
                     node→leftChild→value;
                     break;
         case true:  node→value = TRUE;
                     break;
         case false: node→value = FALSE;
      }
   }
}
```

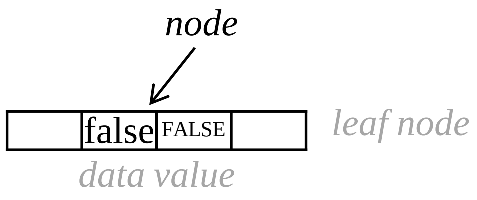
Program 5.9: Postorder evaluation function

not/and/or in the data field of non-leaf nodes

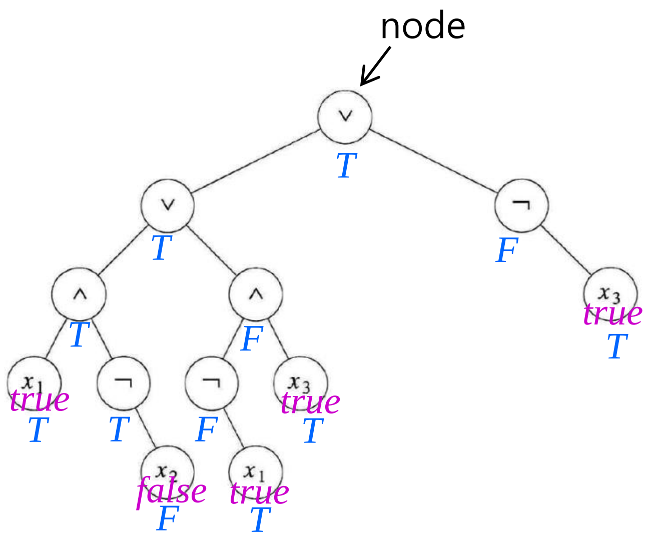
true/false in the data field of leaf nodes, x_1, x_2, and x_3

```
void postOrderEval(treePointer node)
{/* modified post order traversal to evaluate a
    propositional calculus tree */
   if (node) {
      postOrderEval(node→leftChild);   L
      postOrderEval(node→rightChild);  R
      switch(node→data) {              V
        case not:    node→value =
                     !node→rightChild→value;
                     break;
        case and:    node→value =
                     node→rightChild→value &&
                     node→leftChild→value;
                     break;
        case or:     node→value =
                     node→rightChild→value ||
                     node→leftChild→value;
                     break;
        case true:   node→value = TRUE;
                     break;
        case false:  node→value = FALSE;
      }
   }
}
```





```
void postOrderEval(treePointer node)
{/* modified post order traversal to evaluate a
    propositional calculus tree */
   if (node) {
      postOrderEval(node→leftChild);      L
      postOrderEval(node→rightChild);     R
      switch(node→data) {                 V
        case not:    node→value =
              !node→rightChild→value;
              break;
        case and:    node→value =
              node→rightChild→value &&
              node→leftChild→value;
              break;
        case or:     node→value =
              node→rightChild→value ||
              node→leftChild→value;
              break;
        case true:   node→value = TRUE;
              break;
        case false:  node→value = FALSE;
      }
   }
}
```



ex) for a combination

(x_1, x_2, x_3) = (*true*, *false*, *true*)

5.6 Heaps

5.6.1 Priority Queues

- *Priority queues*
  - *deletion*: deletes the element with the highest(or the lowest) priority
  - *insertion* : insert an element with arbitrary priority

  (ex: job scheduling in OS)

- We use *max(min) heap* to implement the priority queues

ADT *MaxPriorityQueue* is

objects: a collection of $n > 0$ elements, each element has a key

functions:

for all $q \in$ *MaxPriorityQueue*, *item* $\in$ *Element*, $n \in$ integer

MaxPriorityQueue create(*max_size*)	::=	create an empty priority queue.
Boolean isEmpty(q, n)	::=	**if** ($n > 0$) **return** *FALSE* **else return** *TRUE*
Element top(q, n)	::=	**if** (!isEmpty(q, n)) **return** an instance of the largest element in q **else return** error.
Element pop(q, n)	::=	**if** (!isEmpty(q, n)) **return** an instance of the largest element in q and remove it from the heap **else return** error.
MaxPriorityQueue push(q, *item*, n)	::=	insert *item* into q and return the resulting priority queue.

ADT 5.2: Abstract data type *MaxPriorityQueue*

5.6.2 Definition of a Max Heap

- **Definition :**
  - A ***max tree*** is a tree in which the key value in each node is no smaller than the key values in its children (if any). *parent's key $\geq$ children's keys*
  - A ***max heap*** is a complete binary tree that is also a max tree

- **Definition :**
  - A ***min tree*** is a tree in which the key value in each node is no larger than the key values in its children (if any). *parent's key $\leq$ children's keys*
  - A ***min heap*** is a complete binary tree that is also a min tree.

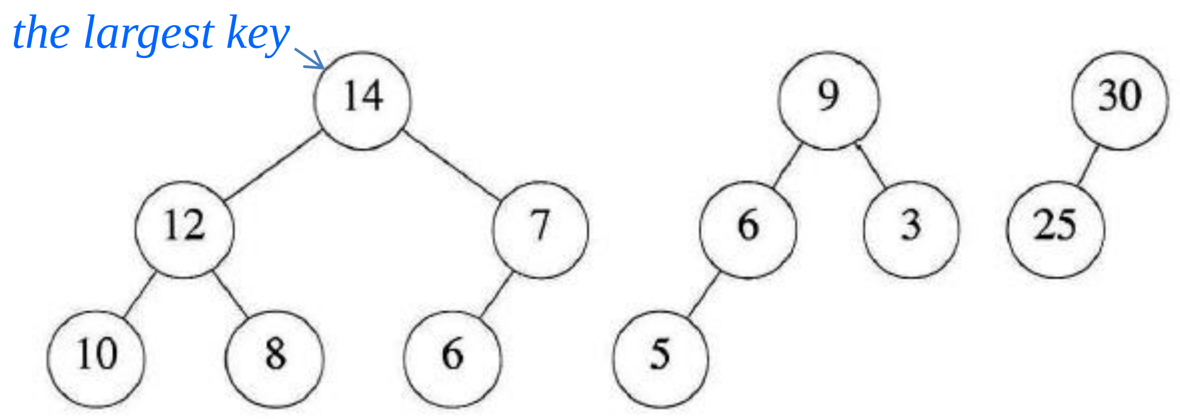


Figure 5.25: Max heaps

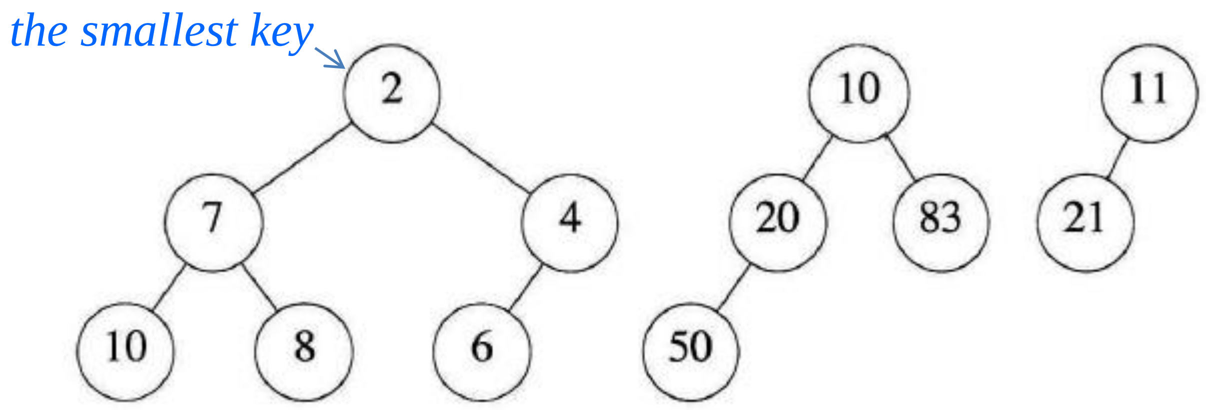


Figure 5.26: Min heaps

5.6.3 Insertion into a Max Heap

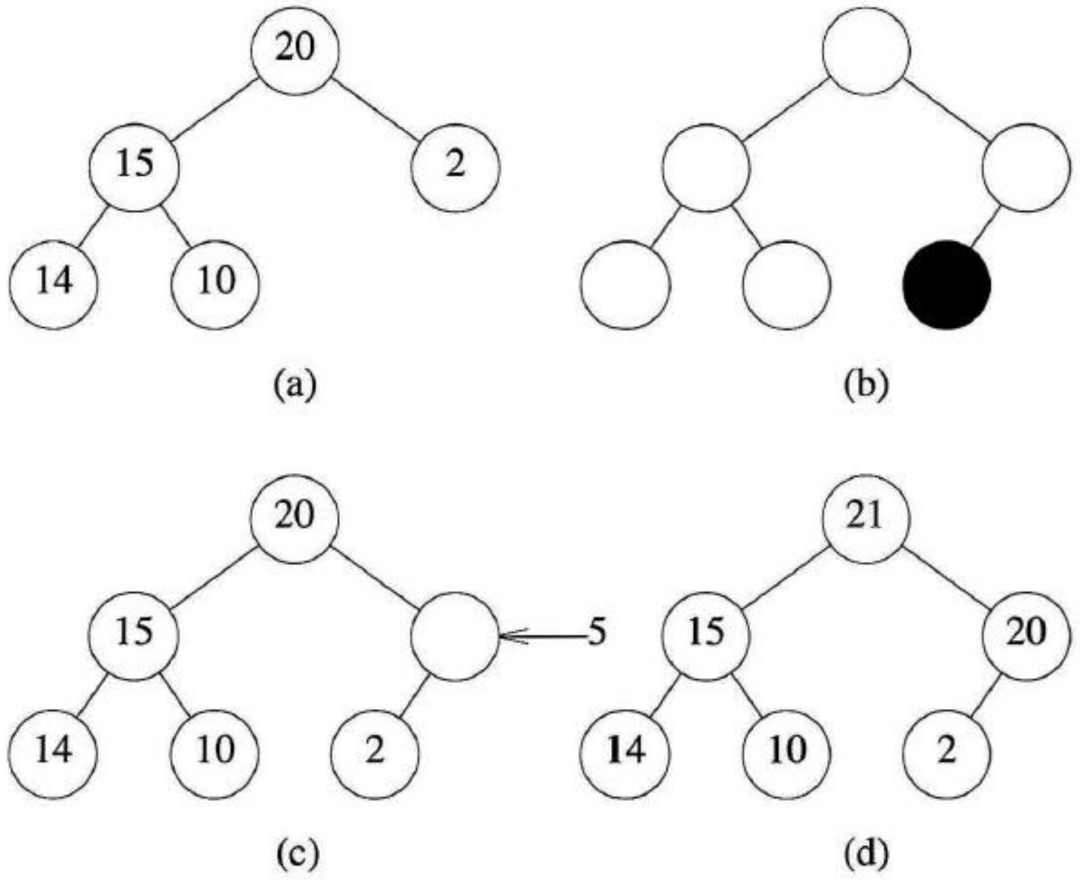


Figure 5.27: Insertion into a max heap

$$\begin{tikzcd}
(a) \arrow[d, "\mathit{Insert(1)/\ Insert(5)/\ Insert(21)}"] \\
?
\end{tikzcd}$$

```
#define MAX_ELEMENTS 200 /* maximum heap size+1 */
#define HEAP_FULL(n) (n == MAX_ELEMENTS-1)
#define HEAP_EMPTY(n) (!n)
typedef struct {
        int key;
        /* other fields */
        } element;
element heap[MAX_ELEMENTS];
int n = 0;
```

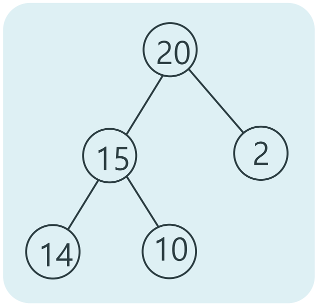
```
void push(element item, int *n)
{/* insert item into a max heap of current size *n */
  int i;
  if (HEAP_FULL(*n)){
    fprintf(stderr, "The heap is full. \n");
    exit(EXIT_FAILURE);
  }
  i = ++(*n);
  while ((i != 1) && (item.key > heap[i/2].key)) {
    heap[i] = heap[i/2];
    i /= 2;
  }
  heap[i] = item;
}
```

Program 5.13: Insertion into a max heap

current size

n 5

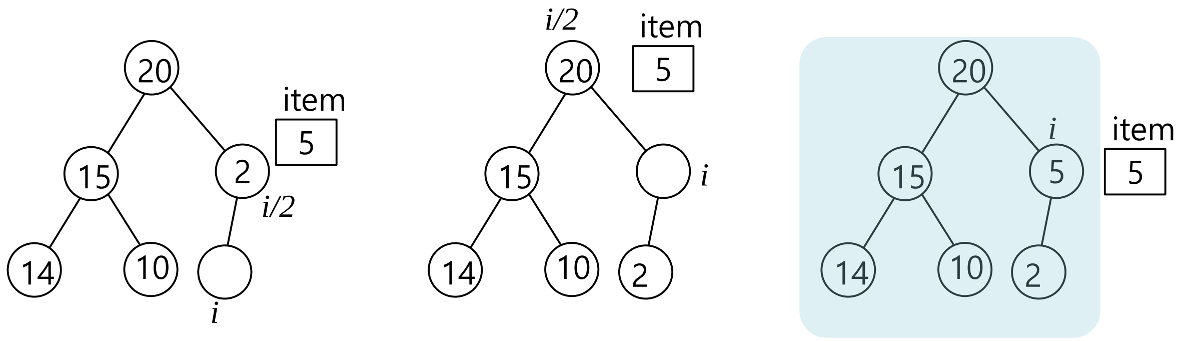
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
heap	-	20	15	2	14	10		...



push(5, &n)

current size

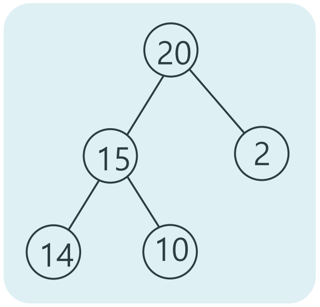
n 6



current size

n 5

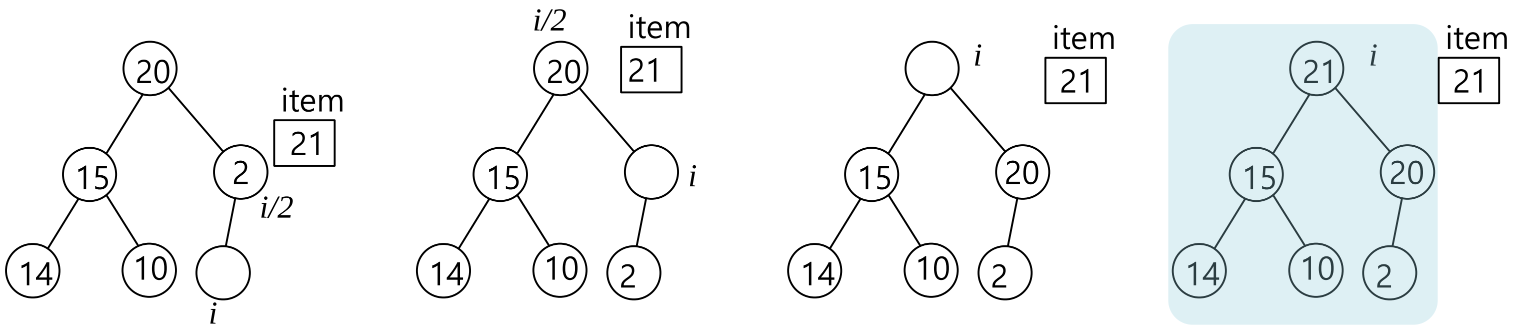
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
heap	-	20	15	2	14	10		...



push(21, &n)

current size

n 6



- Analysis of *push*
  - the height of heap with n elements : $[\log_2(n+1)]$
  - while loop is iterated $O(\log_2 n)$ times
  - time complexity: $O(\log_2 n)$

5.6.4 Deletion from a Max heap

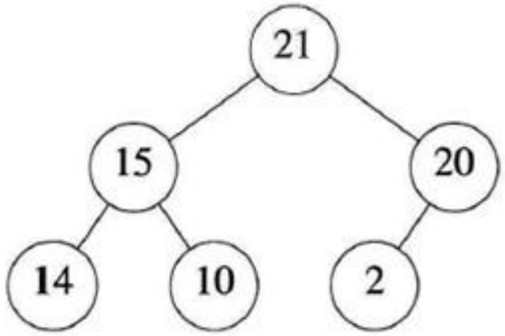


Figure 5.27 (d)

$\xrightarrow{delete(\,)}$

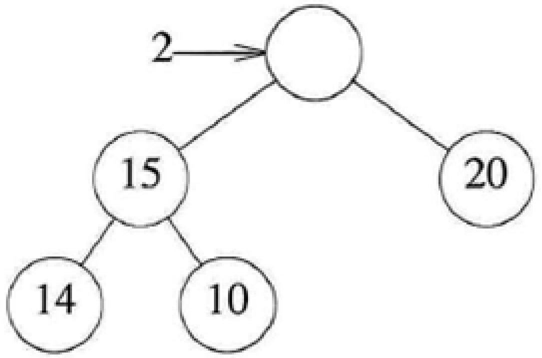


Figure 5.28 (a)

- - - - - - -> ?

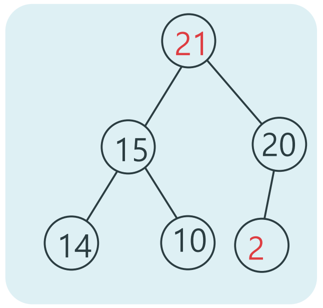
```
element pop(int *n)
{/* delete element with the highest key from the heap */
   int parent, child;
   element item, temp;
   if (HEAP_EMPTY(*n)) {
      fprintf(stderr, "The heap is empty\n");
      exit(EXIT_FAILURE);
   }
   /* save value of the element with the highest key */
   item = heap[1];
   /* use last element in heap to adjust heap */
   temp = heap[(*n)--];
   parent = 1;
   child = 2;
   while (child <= *n) {
      /* find the larger child of the current parent */
      if((child < *n) && (heap[child].key < heap[child+1].key))
         child++;
      if (temp.key >= heap[child].key) break;
      /* move to the next lower level */
      heap[parent] = heap[child];
      parent = child;
      child *= 2;
   }
   heap[parent] = temp;
   return item;
}
```

Program 5.14: Deletion from a max heap

current size

n $\boxed{6}$

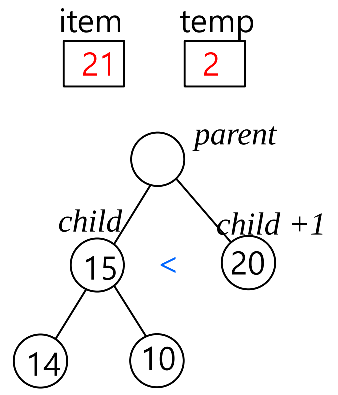
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
heap	-	21	15	20	14	10	2	...

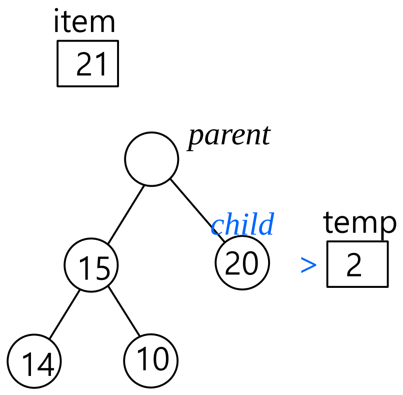


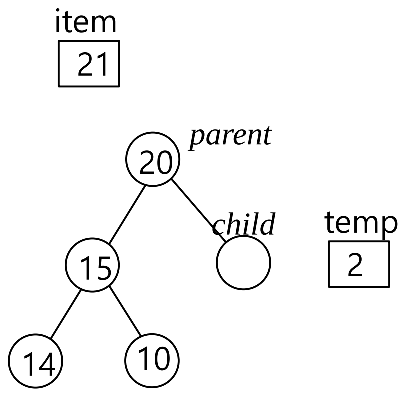
pop(&n)

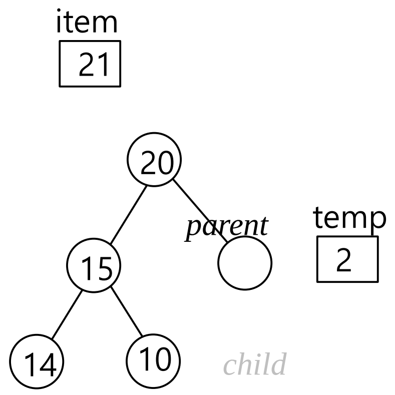
current size

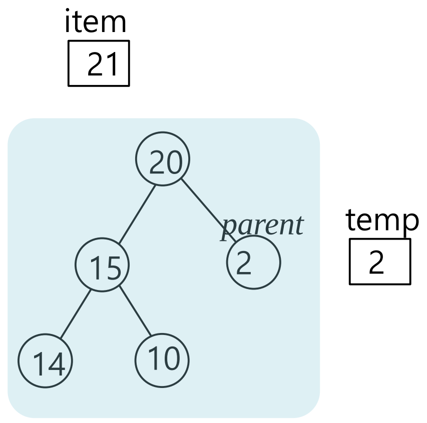
n $\boxed{5}$







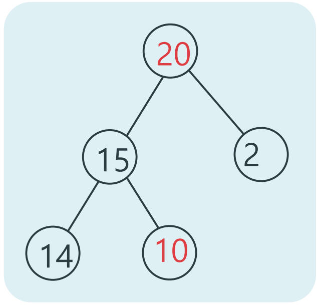




current size

n 5

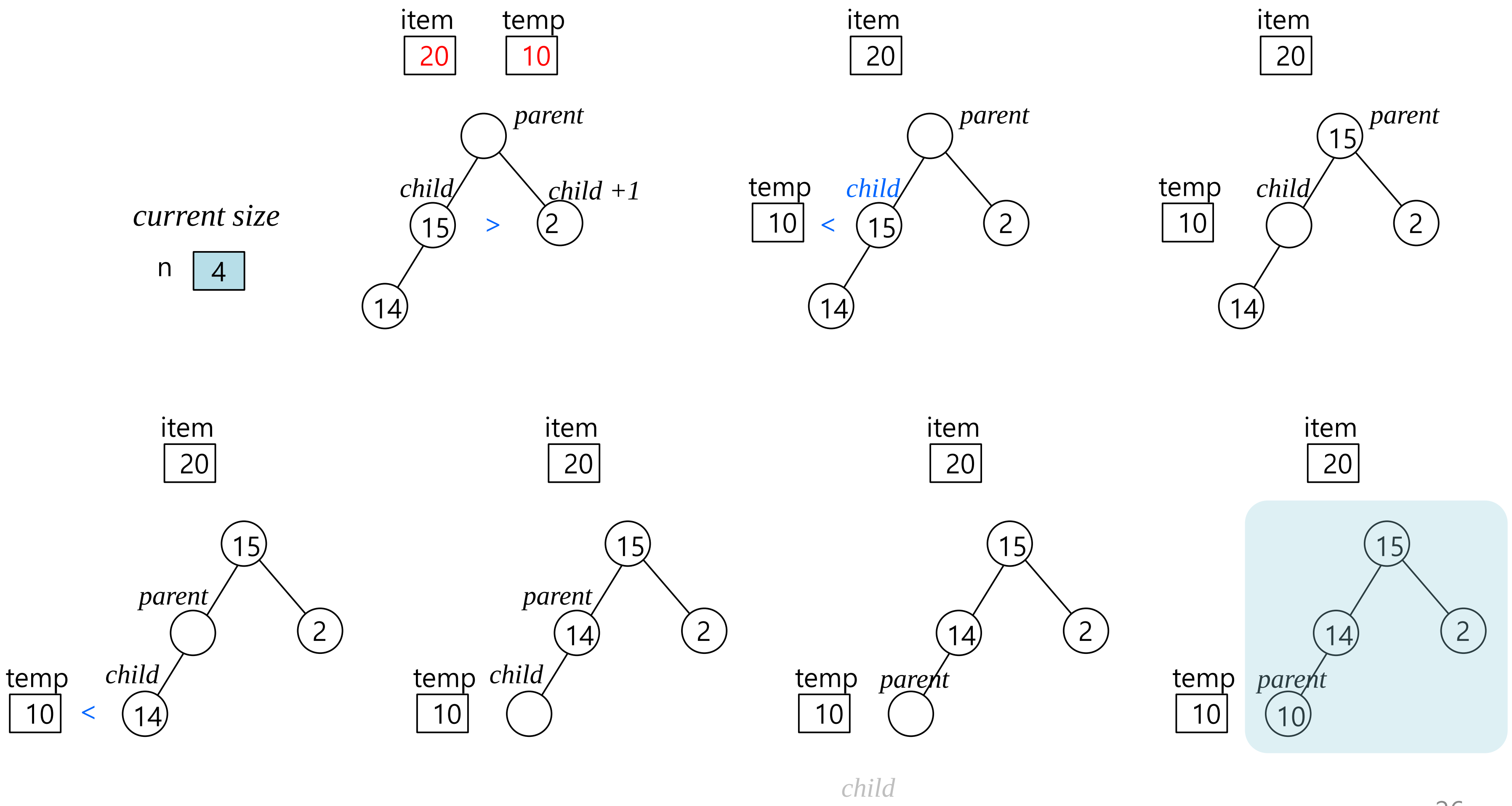
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
heap	-	20	15	2	14	10	...	



pop(&n)

current size

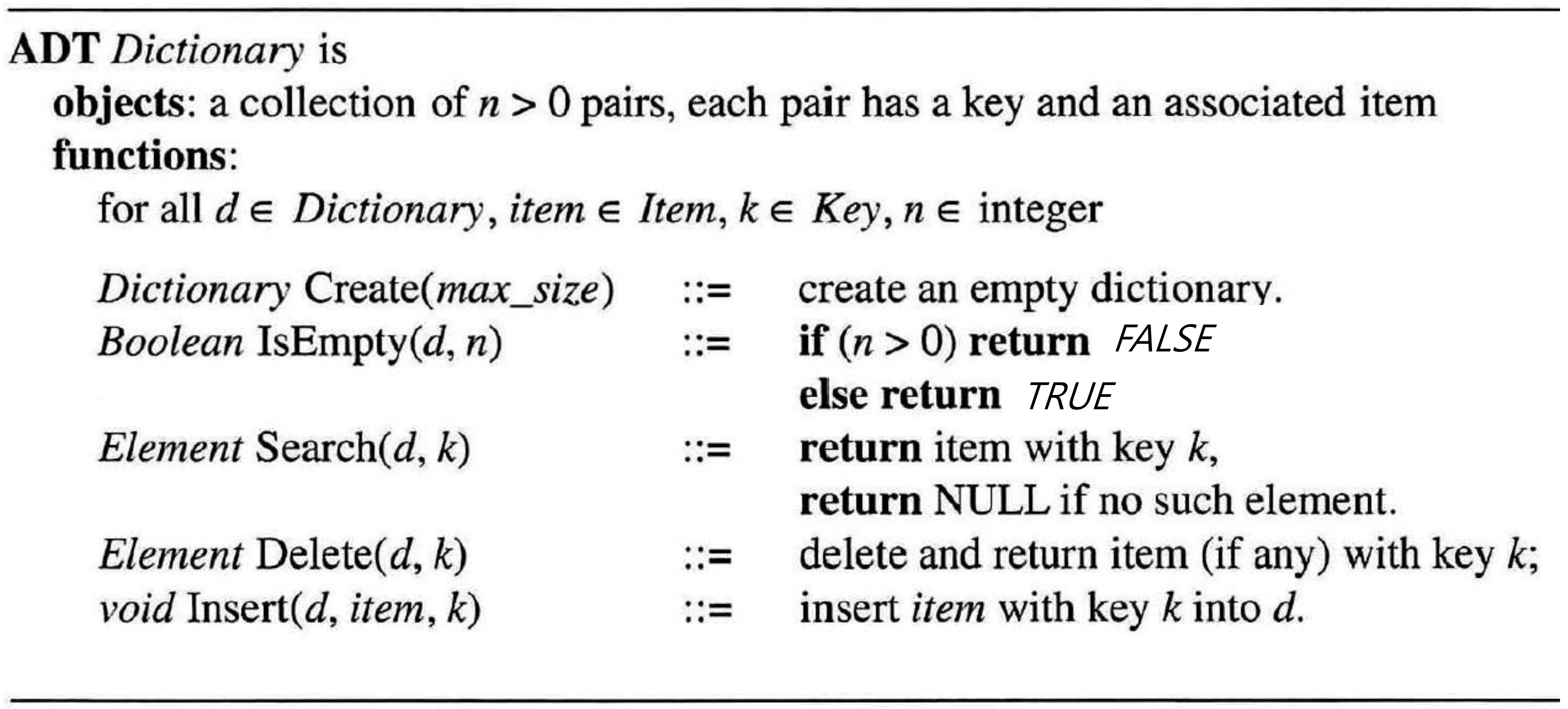
n 4



- Analysis of *pop*
  - the height of heap with n elements : $\lceil \log_2(n+1) \rceil$
  - while loop is iterated $O(\log_2 n)$ times
  - time complexity: $O(\log_2 n)$

5.7 Binary Search Trees

5.7.1 Definition



ADT *Dictionary* is

objects: a collection of $n > 0$ pairs, each pair has a key and an associated item

functions:

for all $d \in$ *Dictionary*, *item* $\in$ *Item*, $k \in$ *Key*, $n \in$ integer

Dictionary Create(*max_size*)	::=	create an empty dictionary.
Boolean IsEmpty(*d*, *n*)	::=	**if** ($n > 0$) **return** *FALSE* **else return** *TRUE*
Element Search(*d*, *k*)	::=	**return** item with key *k*, **return** NULL if no such element.
Element Delete(*d*, *k*)	::=	delete and return item (if any) with key *k*;
void Insert(*d*, *item*, *k*)	::=	insert *item* with key *k* into *d*.

ADT 5.3: Abstract data type *dictionary*

- With a ***binary search tree***, these functions can be performed both
  - By key value : eg. delete the element with key x
  - By rank : eg. delete the fifth smallest element
- search, insert, delete : O(*h*), where *h* is the height of the BST.

Definition: A *binary search tree* is a binary tree. It may be empty. If it is not empty then it satisfies the following properties:

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

(2) The keys (if any) in the left subtree are smaller than the key in the root.

(3) The keys (if any) in the right subtree are larger than the key in the root.

(4) The left and right subtrees are also binary search trees. □

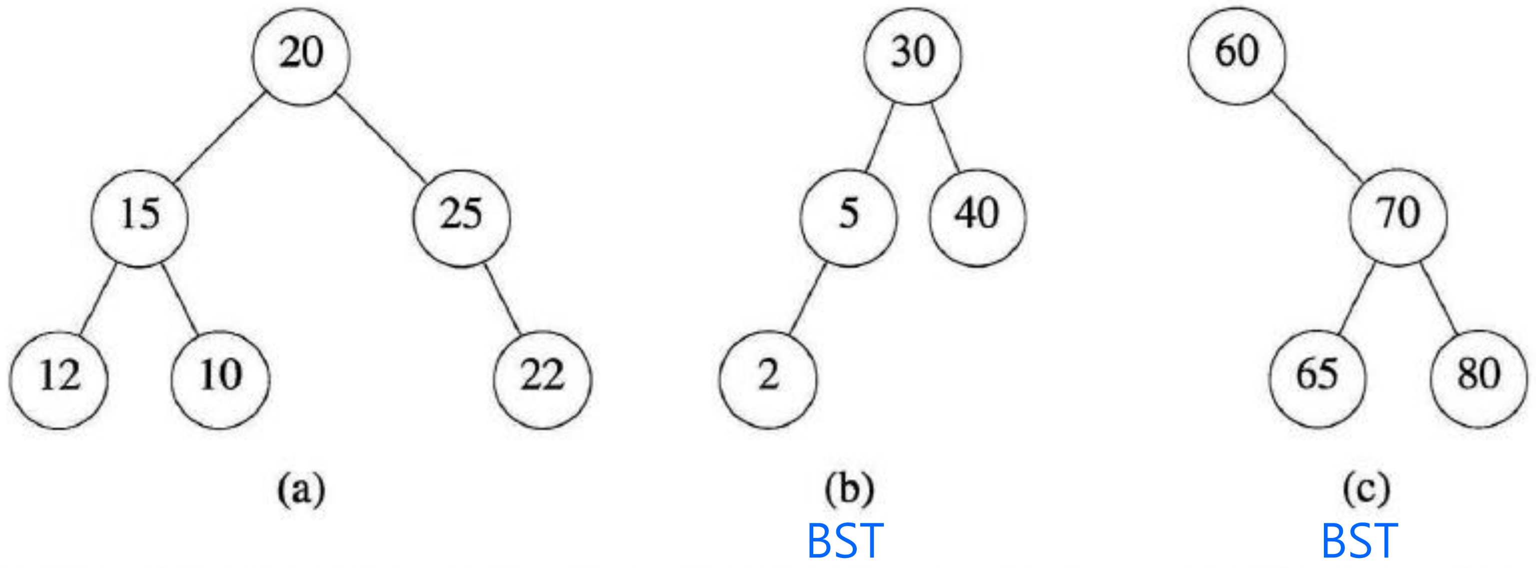


Figure 5.29: Binary trees

5.7.2 Searching a Binary Search Tree

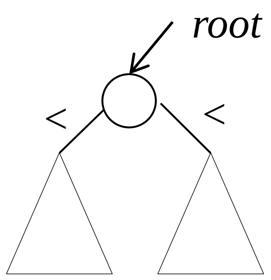
```
typedef int iType;
typedef struct{
      int key;
      iType item;
}element;

typedef struct node *treePointer;
typedef struct node{
      element data;
      treePointer leftChild, rightChild;
}node;
```

```
element* search(treePointer root, int k  )
{/* return a pointer to the element whose key is k, if
    there is no such element, return NULL. */
①  if (!root) return NULL;
②  if (k == root→data.key) return &(root→data);
③  if (k < root→data.key)
      return search(root→leftChild, k);
④  return search(root→rightChild, k);
}
```

Program 5.15: Recursive search of a binary search tree

① the search is unsuccessful
② the search terminates successfully
③ search the left subtree of the root
④ search the right subtree of the root

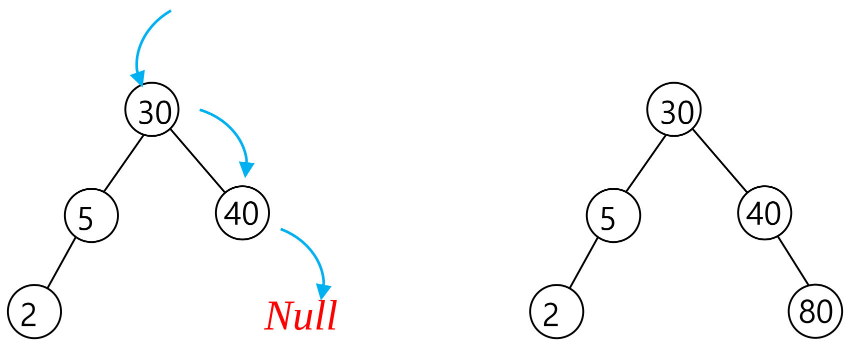


```
element* iterSearch(treePointer tree, int k)
{/* return a pointer to the element whose key is k,  if
    there is no such element, return NULL. */
  while (tree) {
    if (k == tree→data.key) return &(tree→data);
    if (k < tree→data.key)
      tree = tree→leftChild;
    else
      tree = tree→rightChild;
    }
  return NULL;
}
```

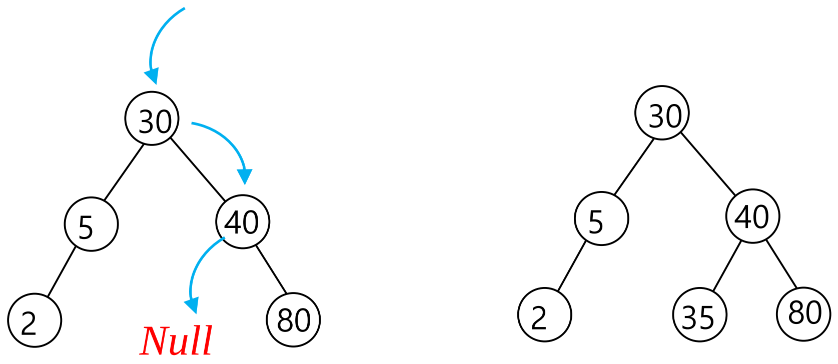
Program 5.16: Iterative search of a binary search tree

- Time complexity of *search* and *iterSearch*:
  - Average case : O(*h*), where *h* is the height of the BST
  - Worst case : O(*n*) for skewed binary tree

5.7.3 Inserting into a Binary Search Tree



(a) insert 80



(b) insert 35

```
void insert(treePointer *node, int k, iType theItem)
{/* if k is in the tree pointed at by node do nothing;
    otherwise add a new node with data = (k, theItem) */
  treePointer ptr, temp = modifiedSearch(*node, k);
  if①(temp || !(*node)) ②{
    /* k is not in the tree */
    MALLOC(ptr, sizeof(*ptr));
    ptr→data.key = k;
    ptr→data.item = theItem;
    ptr→leftChild = ptr→rightChild = NULL;
    if (*node) /* insert as child of temp */
      if (k < temp→data.key) temp→leftChild = ptr;
      else temp→rightChild = ptr;
    else *node = ptr; /* insert into empty BST */
  }
}
```

Program 5.17: Inserting a dictionary pair into a bi

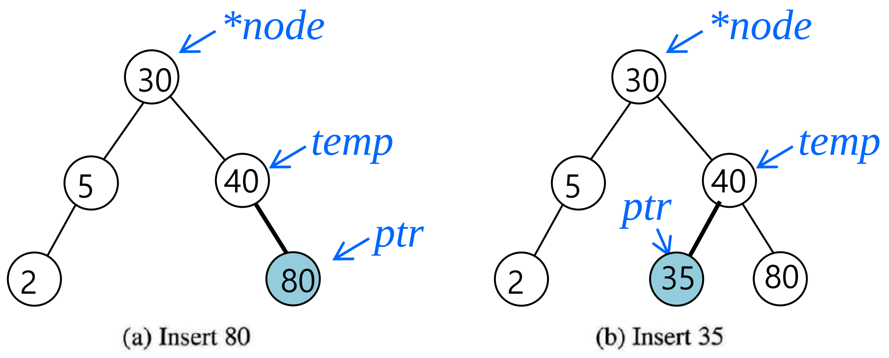
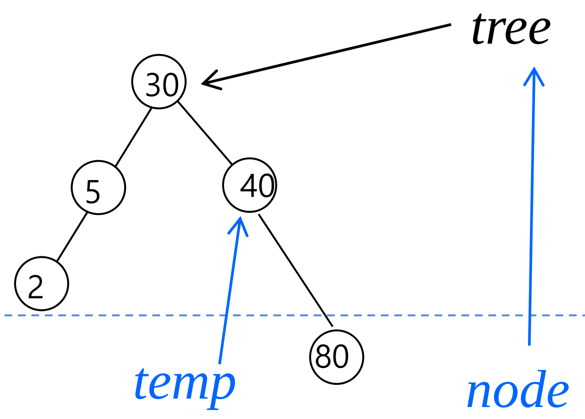


Figure 5.30: Inserting into a binary search tree

- function call ***modifiedSearch(*node, k)***
  - Searches the BST ****node*** for the key ***k***
  - A slightly modified version of *iterSearch*

```
if  the BST is empty or k is present
    return NULL
else
    return the pointer to the last node of the tree
                that was encountered during the search
```

- Analysis of *insert*
  - *modifiedSearch*: O(h), the remaining part : Θ(1)
  - Overall time complexity : O(h)



insert(&tree,80, theItem)

*insert(treePointer *node, int k, iType theItem)*

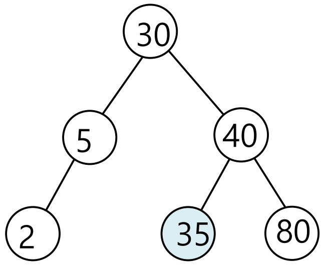
5.7.4 Deletion from a Binary Search Tree

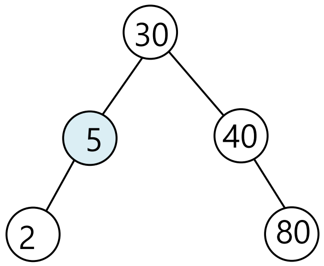
- Deletion from BST

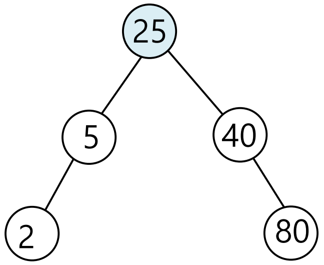
  (1) Deletion of a *leaf* node

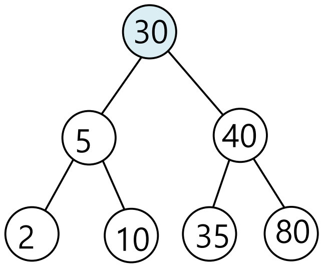
  (2) Deletion of a *nonleaf* node with one child

  (3) Deletion of a *nonleaf* node with two children









Deletion of the node with key 35 ?

5 ?

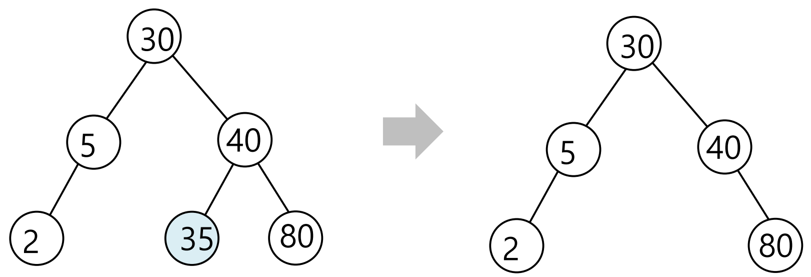
25 ?

30 ?

- Time complexity: O(*h*)

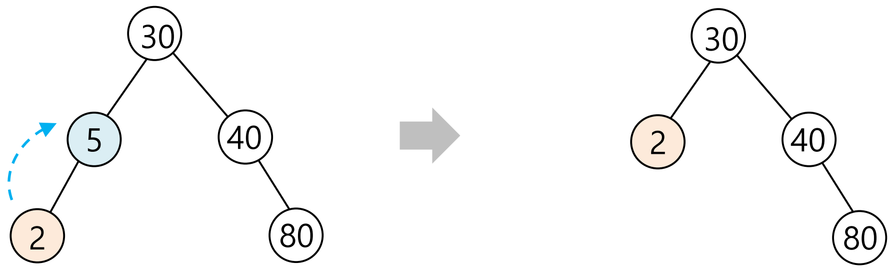
- Deletion from BST

  (1) Deletion of a *leaf* node



(a) delete 35

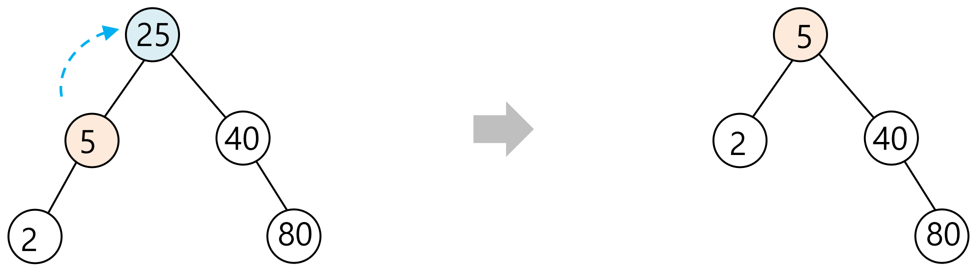
  (2) Deletion of a *nonleaf* node with one child



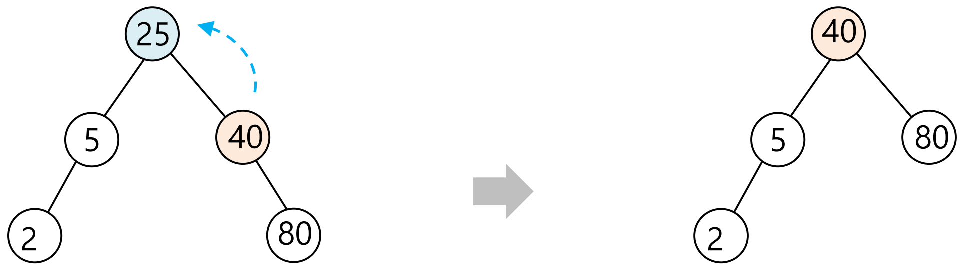
(b) delete 5

- Deletion from BST

(3) Deletion of a *nonleaf* node with two children

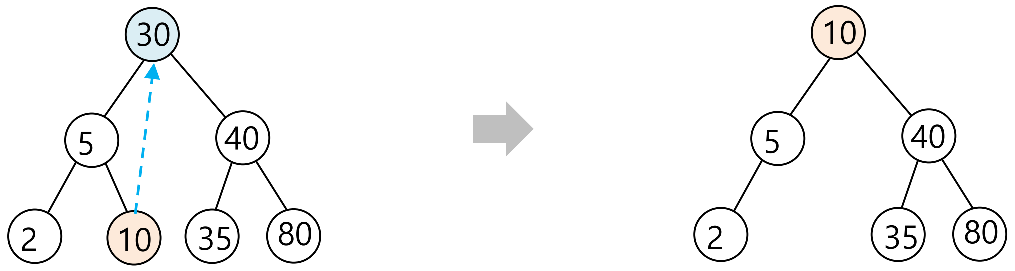


or

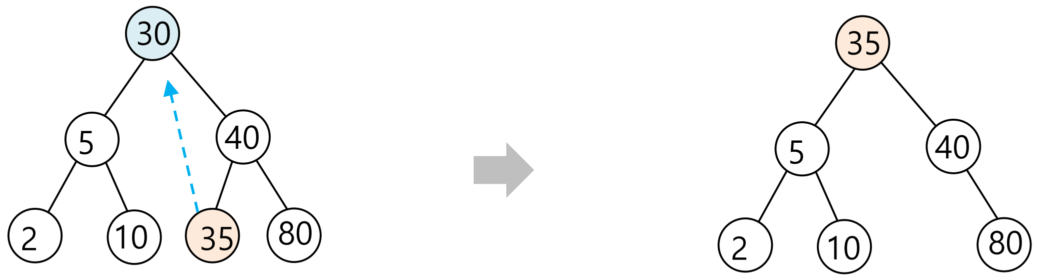


(c) delete 25

- Deletion from BST
  - (3) Deletion of a *nonleaf* node with two children

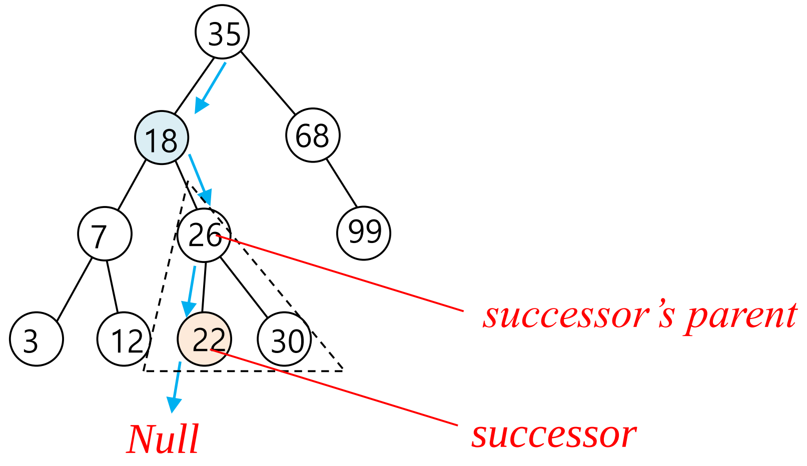


or



(d) delete 30

- Deletion from BST
  - (3) Deletion of a *nonleaf* node with two children
    - how to find the successor node



(e) delete 18

5.7.5 Height of a Binary Search Tree

- The height of a BST can become as large as *n*.
  - $O(\log_2 n)$ on average
  - $O(n)$ on the worst case.

- Balanced Search Trees
  - Worst case height : $O(\log_2 n)$
  - Searching, insertion, or deletion is bounded by $O(h)$, where *h* is the height of a binary tree
  - Ex) AVL tree, 2-3 tree, red-black tree