Chap 5. Trees (5)

Contents

5.1 Introduction

5.2 Binary Trees

5.3 Binary Trees Traversals

5.4 Additional Binary Tree Operations

5.5 Threaded Binary Trees

5.6 Heaps

5.7 Binary Search Trees

5.8 Selection Trees

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** *TRUE* **else return** *FALSE*
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*

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. □

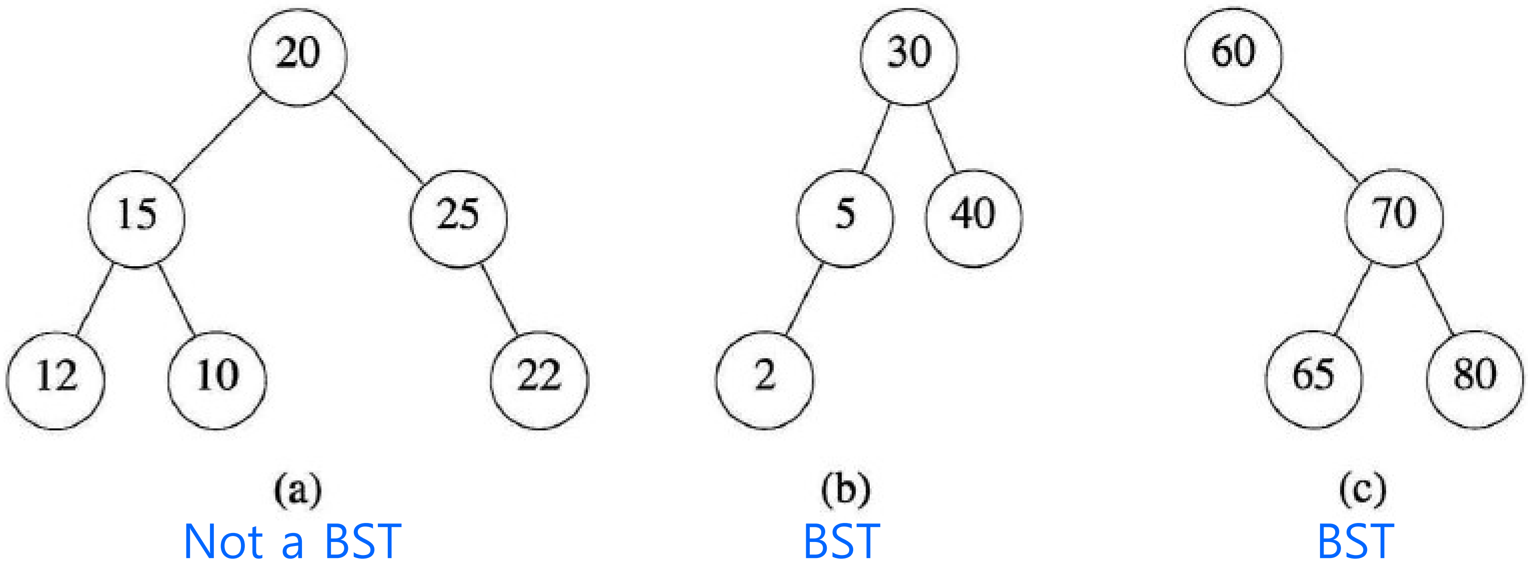


Figure 5.29: Binary trees

5.7.2 Searching a Binary Search Tree

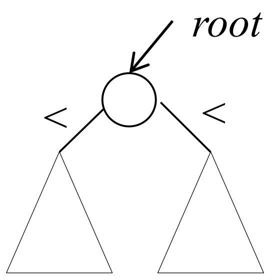
```
typedef int iType;
typedef struct{
      int key;
      iType item;
}element;

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

```
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

```
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);
  if (temp || !(*node)) {
    /* 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 binary search tree

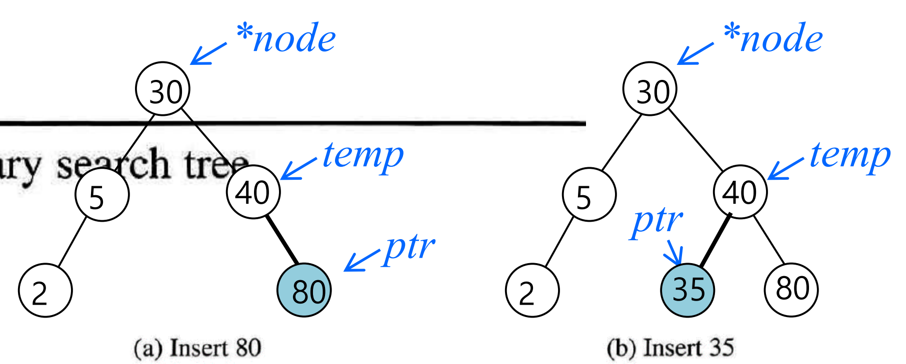


Figure 5.30: Inserting into a binary search tree

- 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 : $\Theta(1)$
  - Overall time complexity : $O(h)$

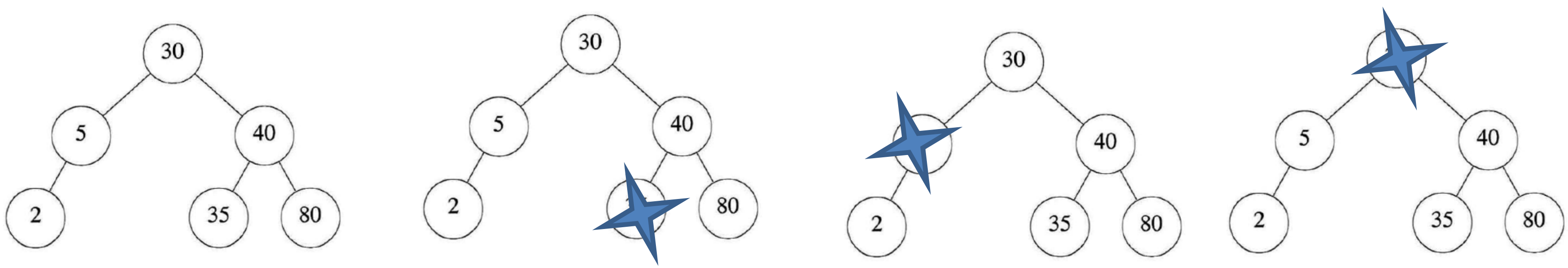
5.7.4 Deletion from a Binary Search Tree

- 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 ? 30 ?

- Time complexity: O(*h*)

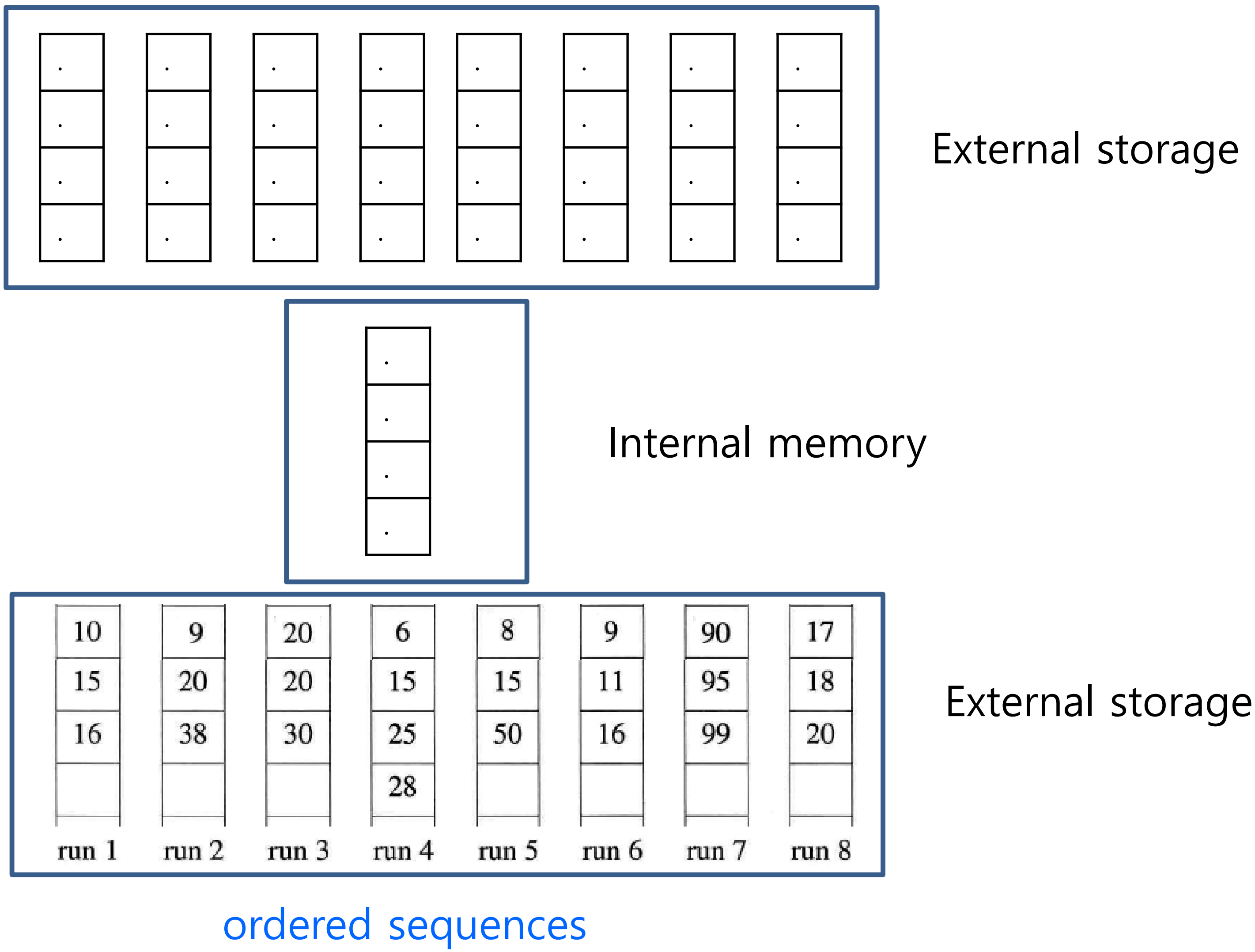
5.7.6 Height of a Binary Search Tree

- 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(**A**delson-**V**elsky and **L**andis) tree, 2-3 tree

5.8 Selection Trees

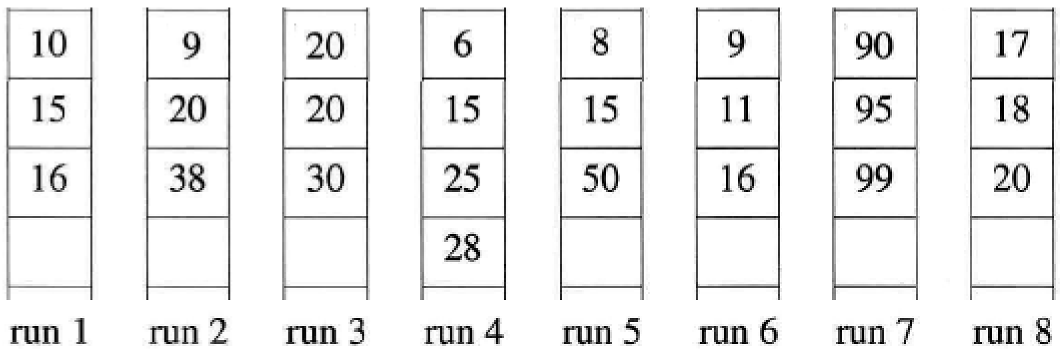
5.8.1 Introduction



5.8 Selection Trees

5.8.1 Introduction

- *k* ordered sequences, called *runs*,
  to be merged into a single ordered sequence.



- The merging task can be accomplished by repeatedly outputting the record with the smallest key.

- For *k*>2, we can *reduce the number of comparisons* by using the **selection tree**; winner trees and loser trees.

5.8.2 Winner Trees

- A *winner tree* is a complete binary tree in which each node represents the smaller of its two children.

root: smallest key

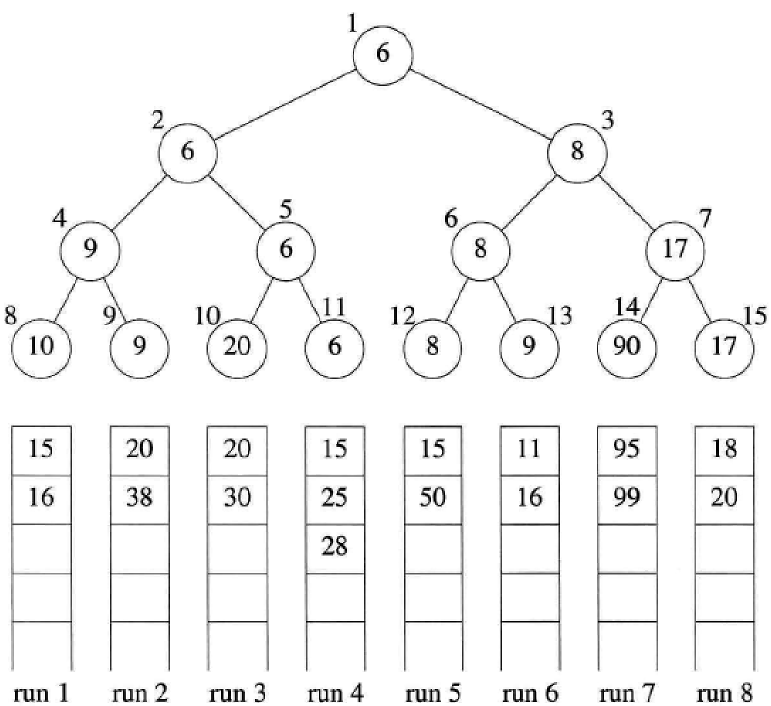
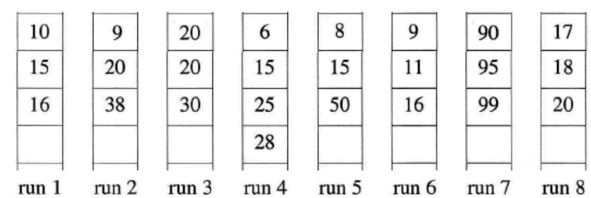


Figure 5.32: Winner tree for $k = 8$, showing the first three keys in each of the eight runs

sequential allocation
(complete binary tree)

Each node contains only a pointer to the record

Leaf node: the first record in the corresponding run



Runs : ordered sequences

implemetation of winner tree

Each node contains only a pointer to the record

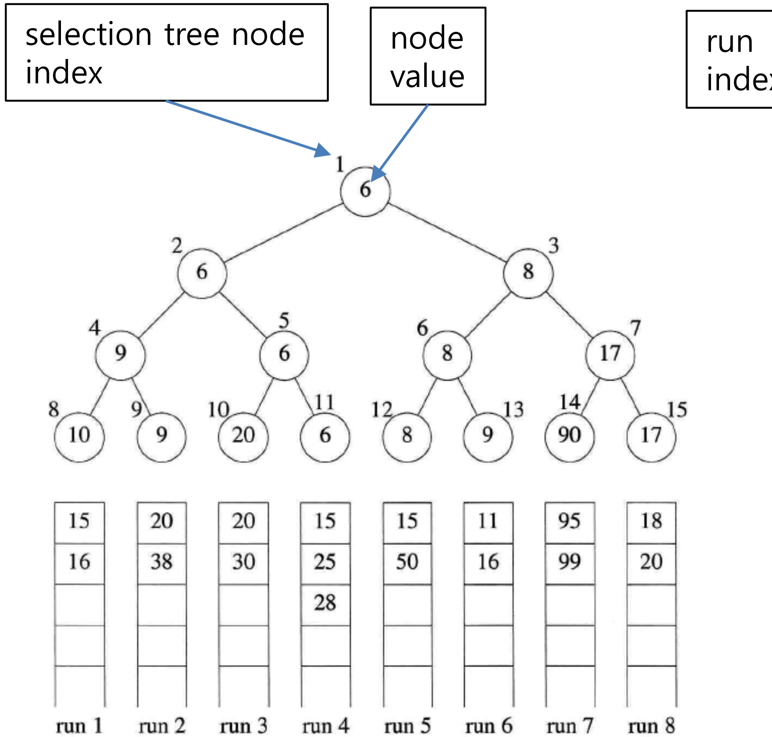
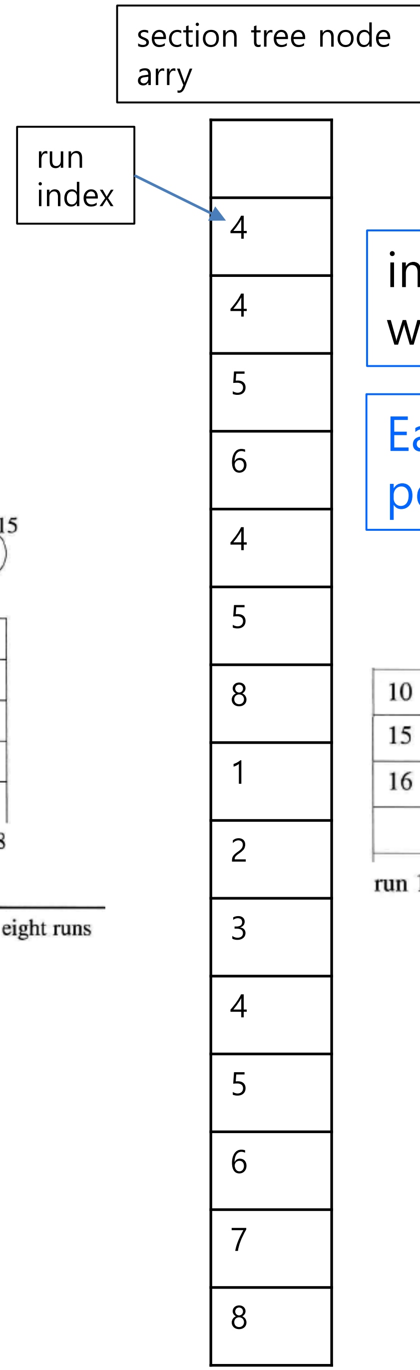
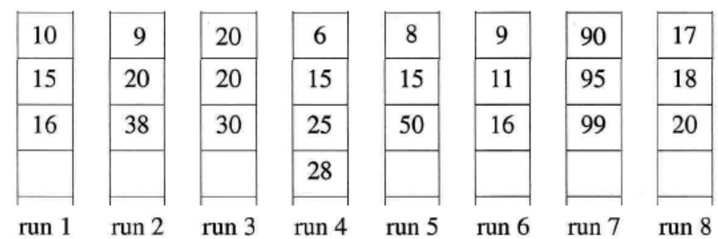
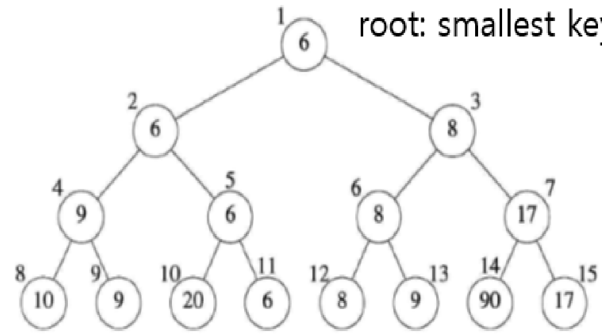


Figure 5.32: Winner tree for $k = 8$, showing the first three keys in each of the eight runs







root: smallest key

	0	1	2	3	4	5	6	7	8
sortedIdx	-	1	1	1	1	1	1	1	1

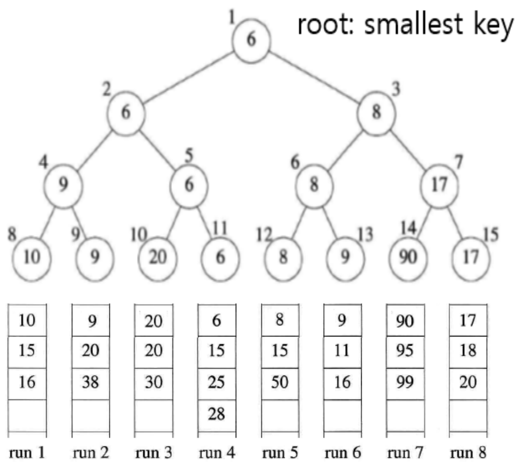
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
winTree	-	4	4	5	2	4	5	8	1	2	3	4	5	6	7	8

nums

각 레코드의 **1**번째 원소들을 정렬해야 함

10	9	20	6	8	9	90	17
11	10	21	7	9	10	91	18
12	11	22	8	10	11	92	19
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
19	18	29	15	17	18	99	26

4
4
5
6
4
5
8
1
2
3
4
5
6
7
8



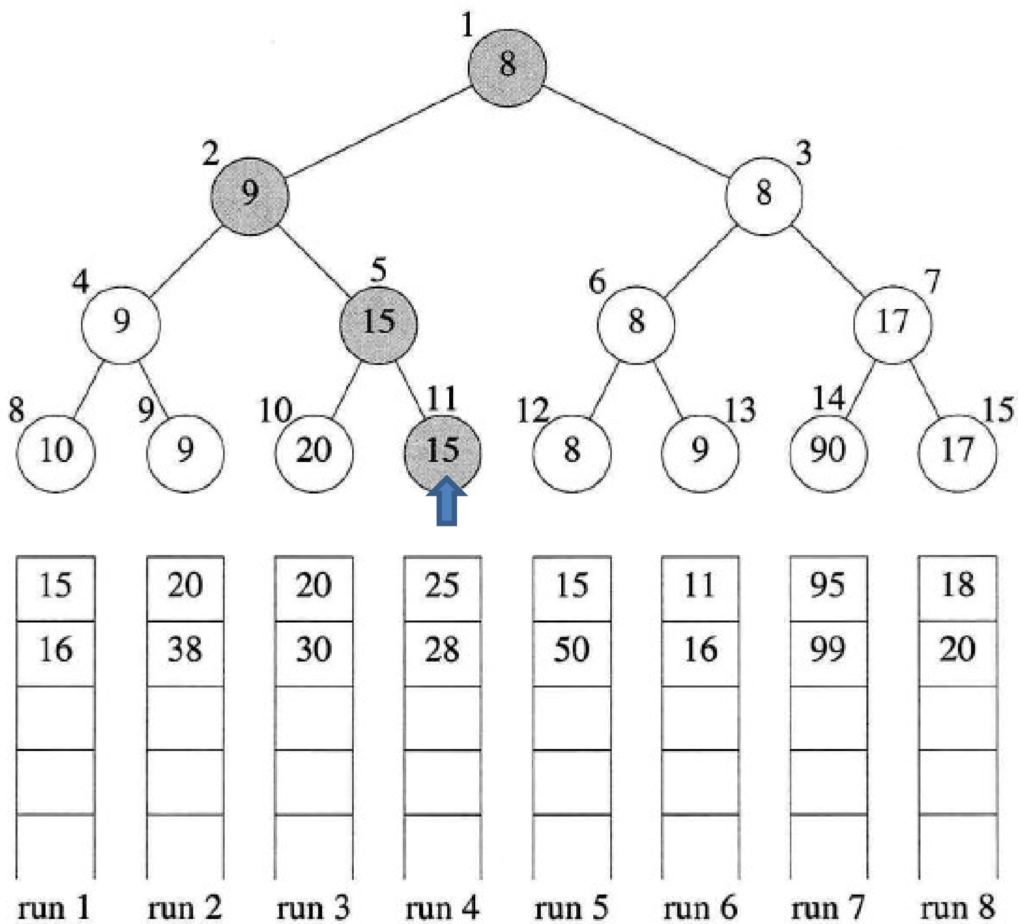
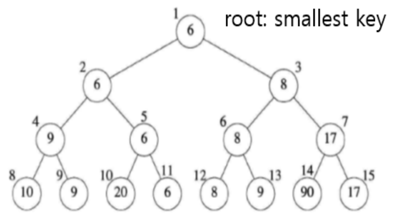


Figure 5.33: Winner tree of Figure 5.32 after one record has been output and the tree restructured (nodes that were changed are shaded)

- Analysis of merging runs using winner trees
  - *Let n be the number of records in all k* runs.
  - The number of levels in the tree is $\lceil \log_2 k + 1 \rceil$
  - The time to restructure the tree is $O(\log_2 k)$.
  - The time required to merge all n records is $O(n\log_2 k)$.
  - The time required to set up the selection tree the first time is $O(k)$.
  - The total time needed to merge the k runs is $O(n\log_2 k)$.



5.8.3 Loser Trees

- A selection tree in which each nonleaf node retains a pointer to the loser is called a *loser tree*.

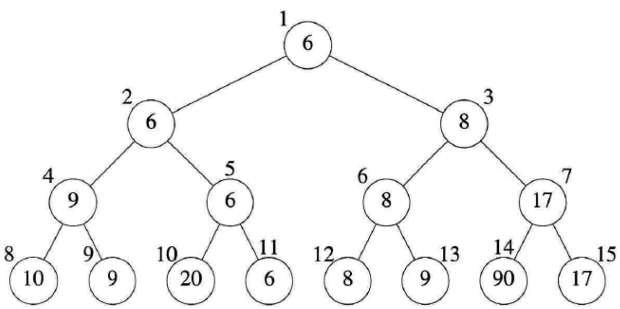
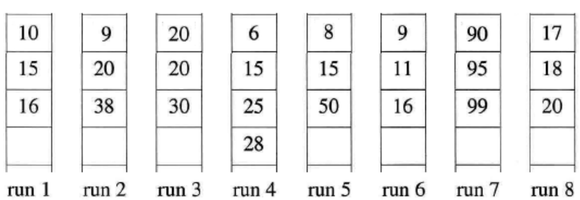


Figure 5.32



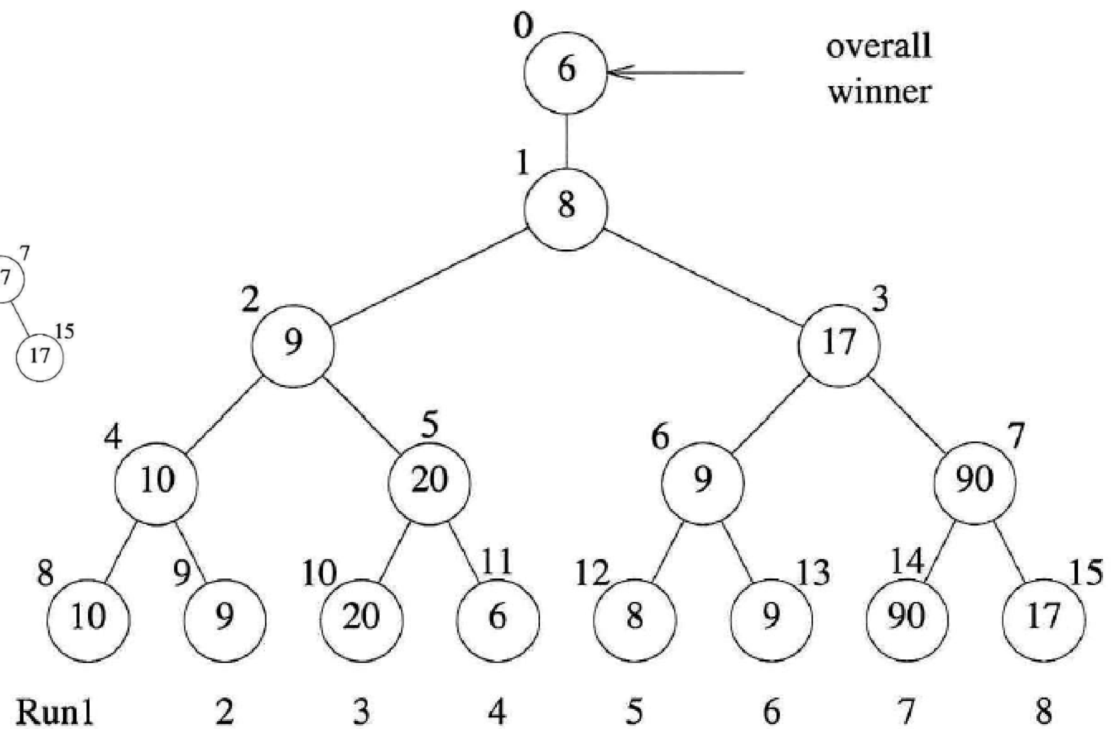


Figure 5.34: Loser tree corresponding to winner tree of Figure 5.32

- In winner tree, following the output of the overall winner, the tree is restructured by playing tournaments along the path from node 11 to node 1.
- *In loser, the records with which the tournaments are to be played* are readily available *from the parent nodes*.
  - As a result, *sibling nodes along the path from 11 to l are not accessed.*