Chap 5. Trees (4)

Contents

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**

5.6 Heaps

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).
- A ***min heap*** is a complete binary tree that is also a min tree.
  parent's key $\leq$ *children's keys*

5.6.2 Definition of a Max Heap

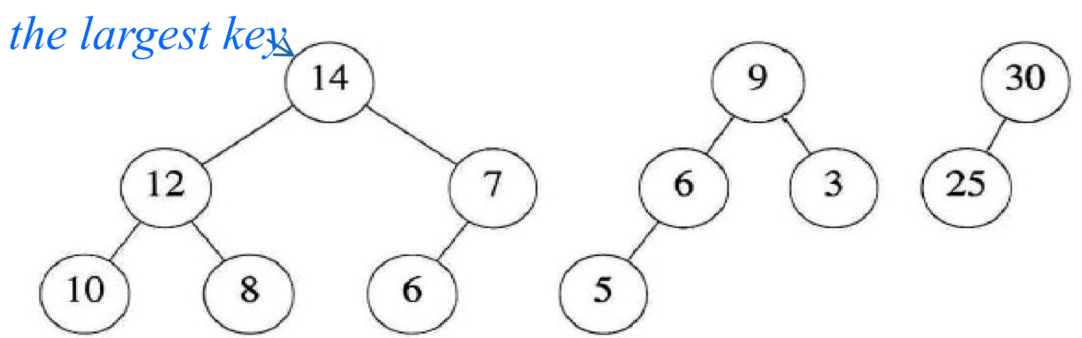


Figure 5.25: Max heaps

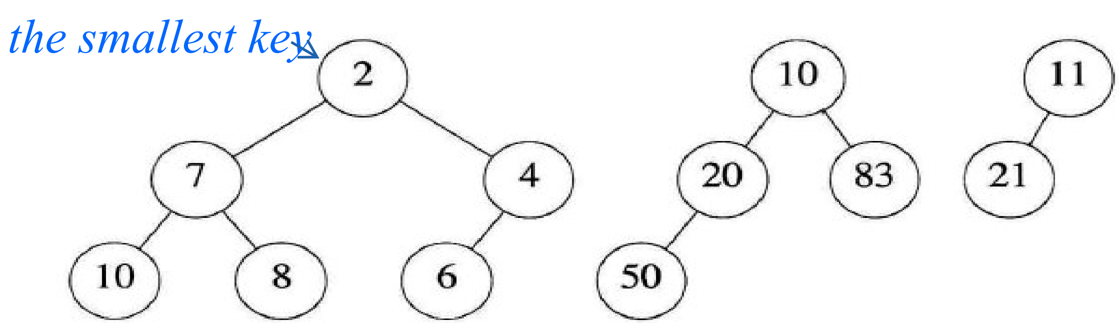


Figure 5.26: Min heaps

5.6.3 Insertion into a Max Heap

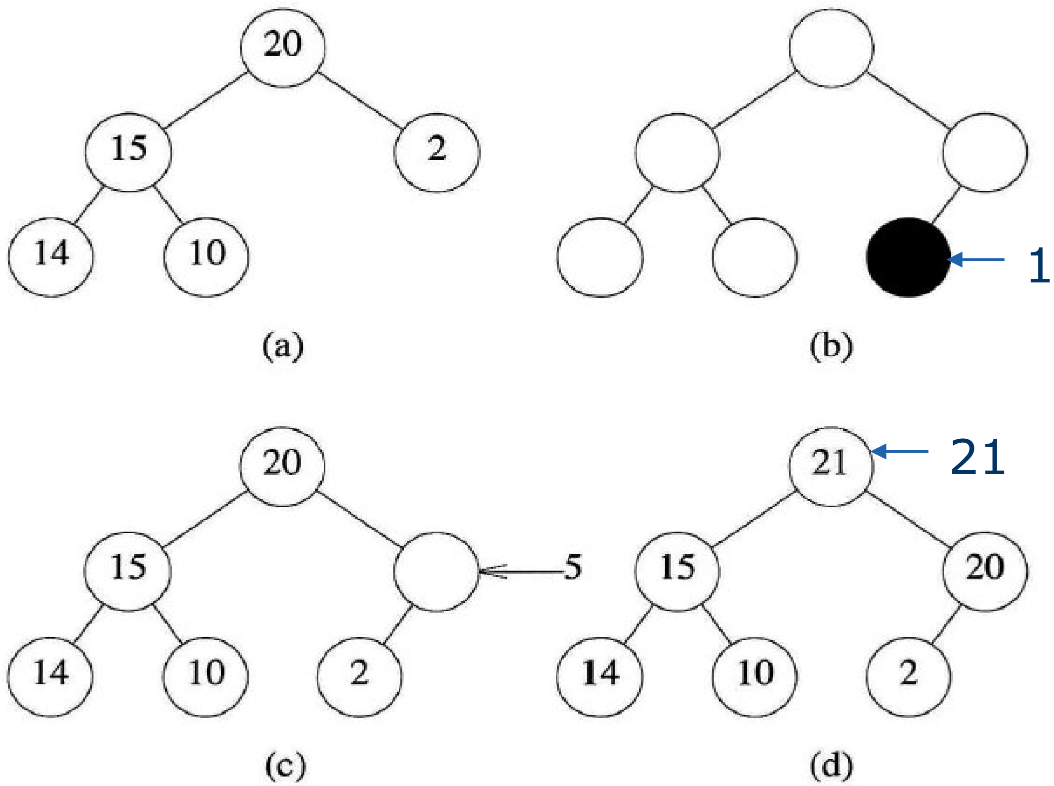


Figure 5.27: Insertion into a max heap

Insert(1)/
Insert(5)/
Insert(21)

5.6.3 Insertion into a Max Heap

```
#define MAX-ELEMENTS 200 /* maximum heap size+l */
#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;
```

5.6.3 Insertion into a Max Heap

```
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

5.6.2 Definition of a Max Heap

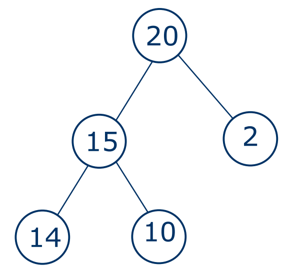
```
void push(element item, int *n)
{
    i = ++(*n);
    while ((i != 1) && (item.key > heap[i/2].key)) {
        heap[i] = heap[i/2];
        i /= 2;
    }
    heap[i] = item;
}
```

current size

n: 5

heap

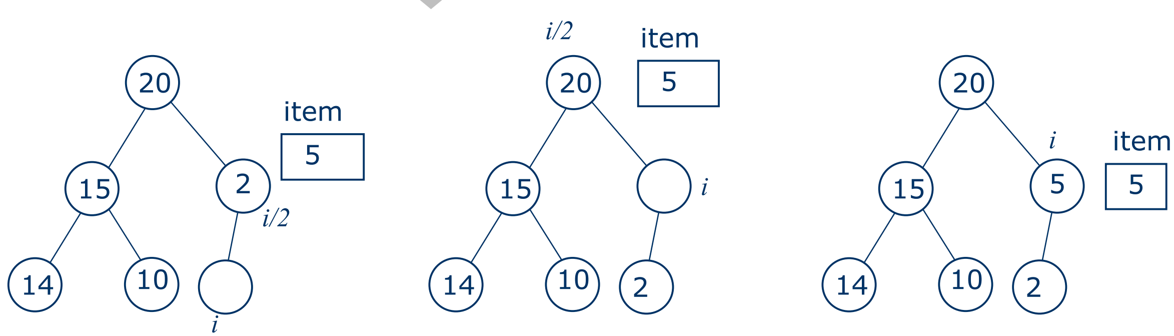
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
-	20	15	2	14	10		...



push(5, &n)

current size

n: 6



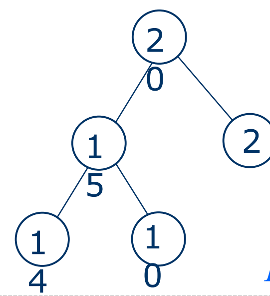
5.6.2 Definition of a Max Heap

```
void push(element item, int *n)
{
    ...
   i = ++(*n);
   while ((i != 1) && (item.key > heap[i/2].key)) {
      heap[i] = heap[i/2];
      i /= 2;
   }
   heap[i] = item;
}
```

current size

n: 5

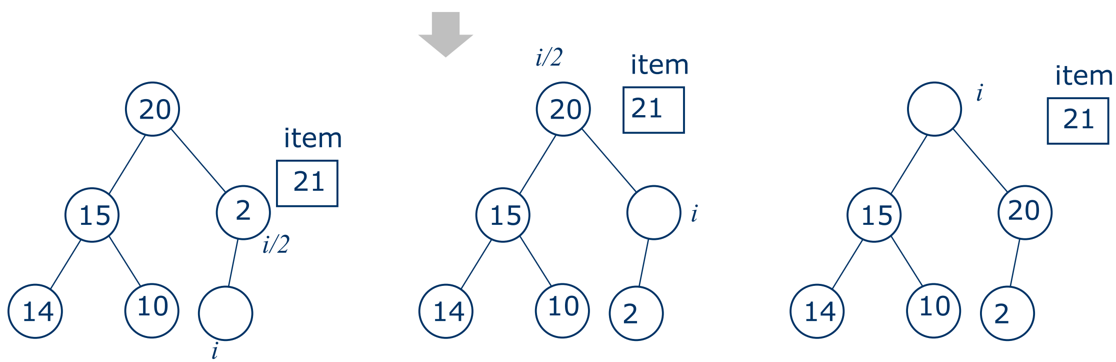
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
heap	-	20	15	2	14	10		...



push(21, &n)

current size

n: 6



5.6.2 Definition of a Max Heap

Analysis of *push*

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

5.6.4 Deletion from a Max heap

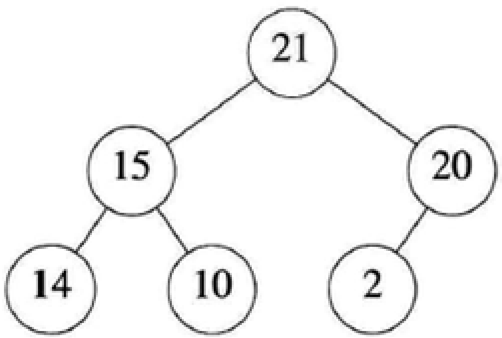


Figure 5.27 (d)

delete( )

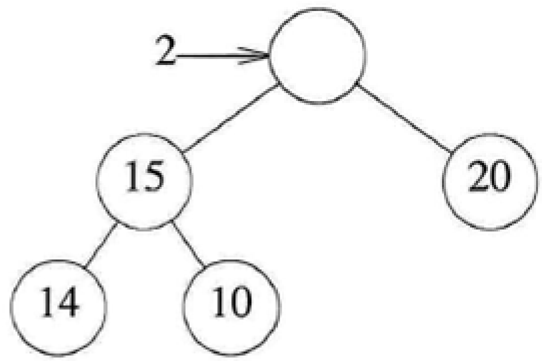
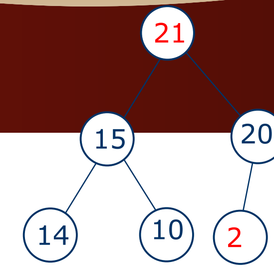


Figure 5.28 (a)

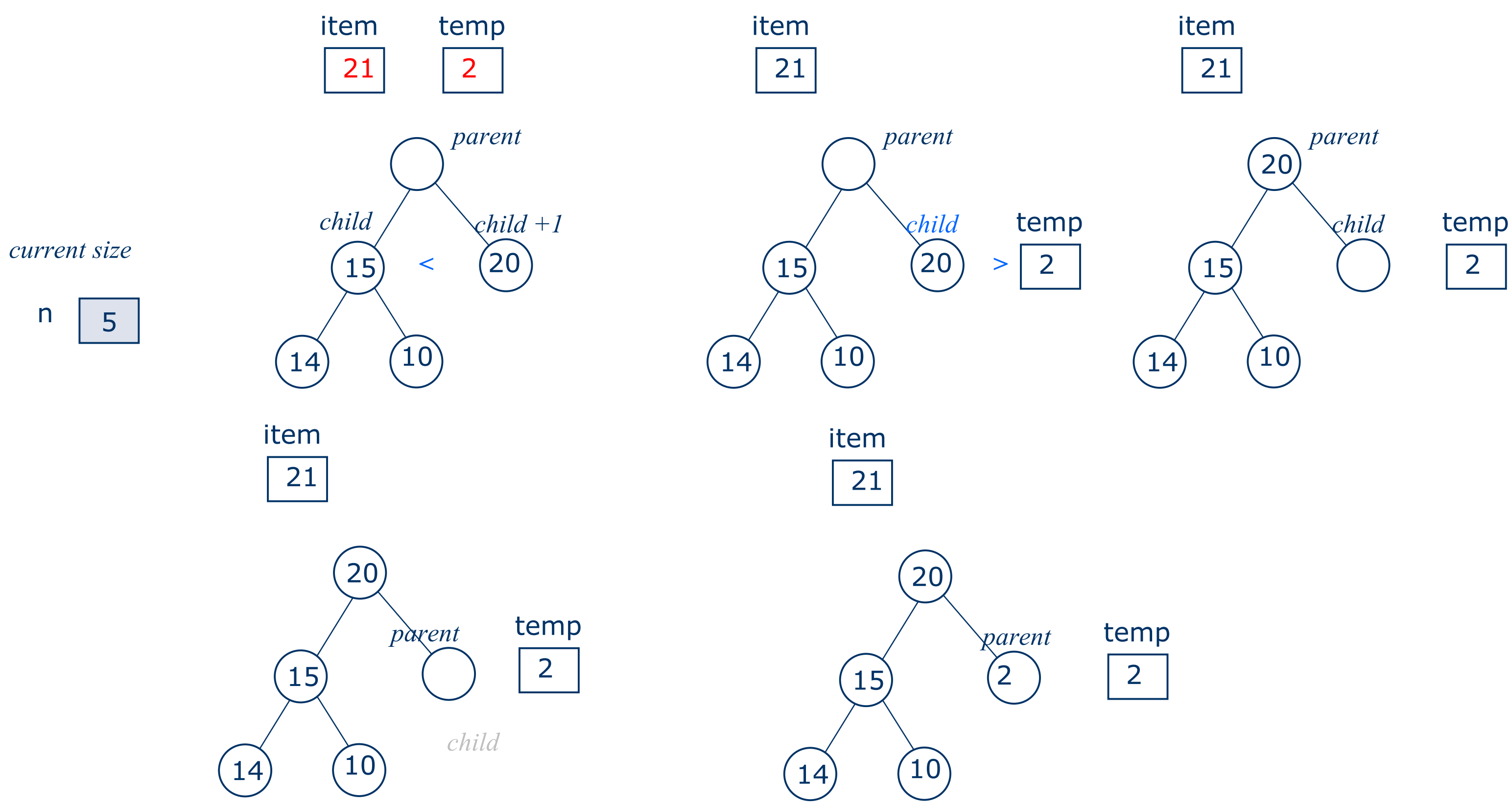
current size

n 6

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



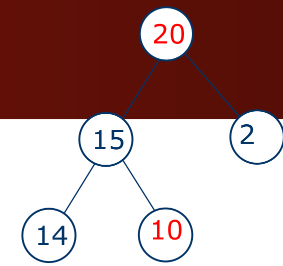
pop(&n)



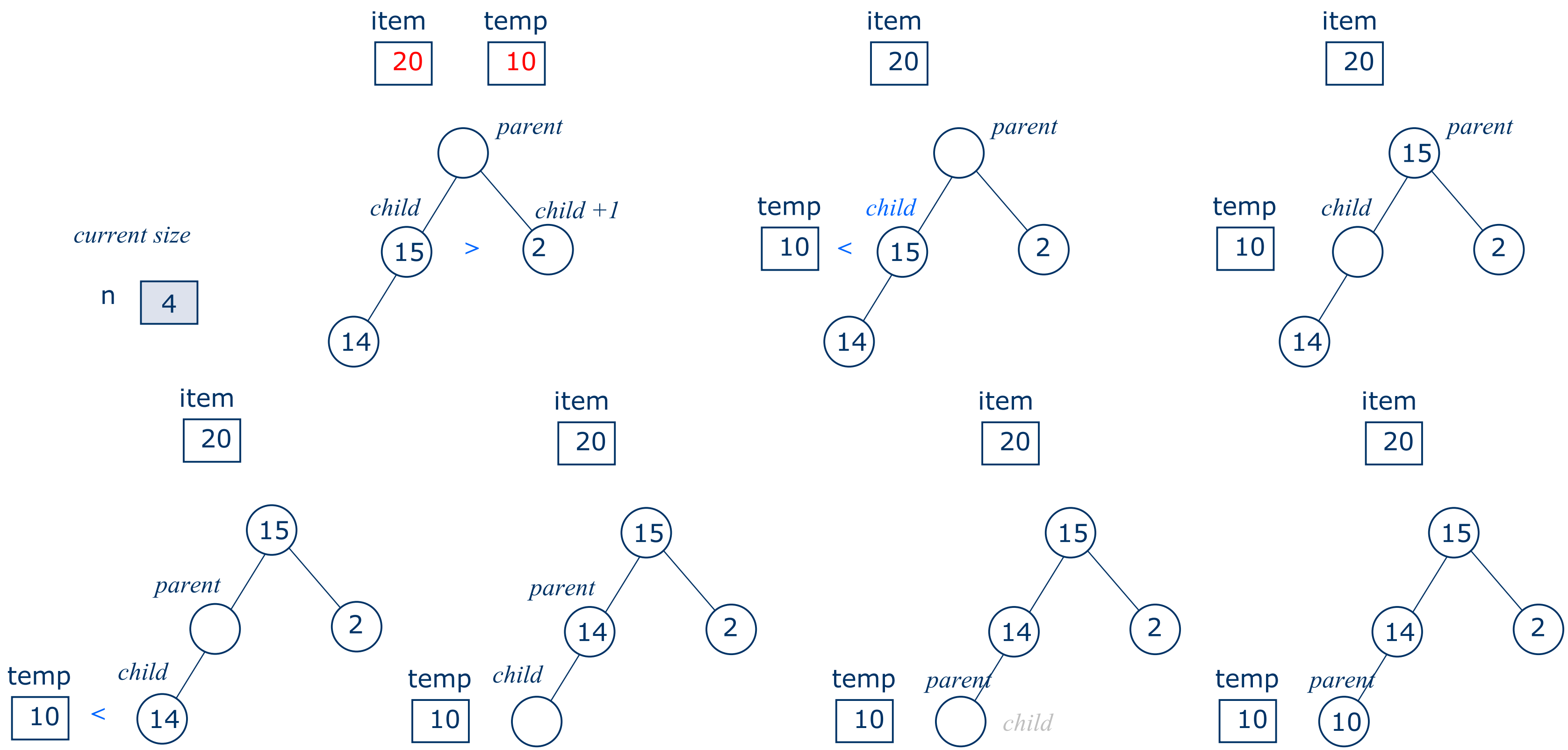
current size

n 5

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



pop(&n)



```
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

5.6.4 Deletion from a Max heap

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

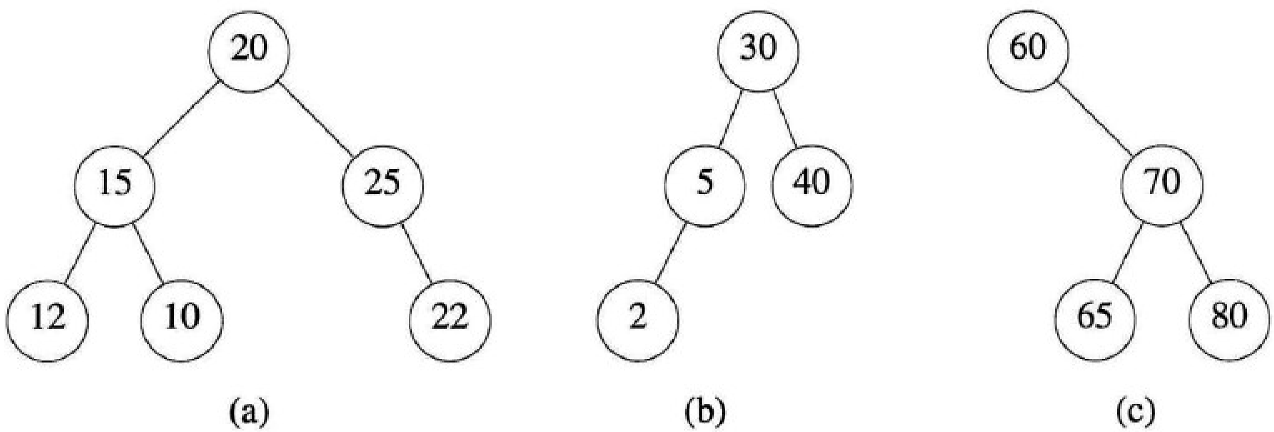
Dictionary Create(*max_size*)	::=	create an empty dictionary.
Boolean IsEmpty(d, n)	::=	**if** ($n > 0$) **return** *TRUE* **else return** *FALSE*
Element Search(d, k)	::=	**return** item with key k, **return** NULL if no such element.
Element Delete(d, k)	::=	delete and return item (if any) with key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

ADT 5.3: Abstract data type *dictionary*

5.7 Binary Search Trees

- **Definition: A binary search tree is a binary tree. It may be empty. If it is not empty then it satisfies the following properties:**
  - Each node has exactly one key and the keys in the tree are distinct.
  - The keys (if any) in the left subtree are smaller than the key in the root.
  - The keys (if any) in the right subtree are larger than the key in the root.
  - The left and right subtrees are also binary search trees.

5.7 Binary Search Trees



Not a BST　　BST　　BST

Figure 5.29: Binary trees

5.7.2 Searching a Binary Search Tree

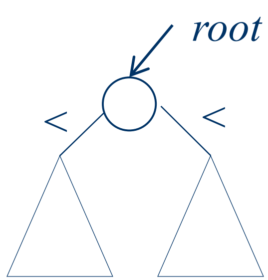
```
typedef int iType;
typedef struct{
      int key;
      iType item;
}element;

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

5.7.2 Searching a Binary Search Tree

```
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



5.7.2 Searching a Binary Search Tree

```
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

5.7.2 Searching 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;
    }
}
```

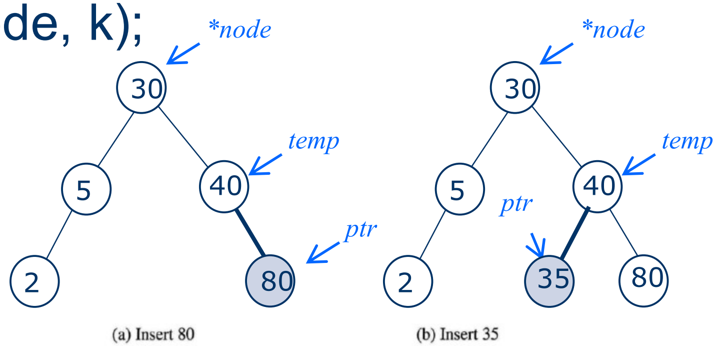


Figure 5.30: Inserting into a binary search tree

Program 5.17: Inserting a dictionary pair into a binary search tree

5.7.3 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*)

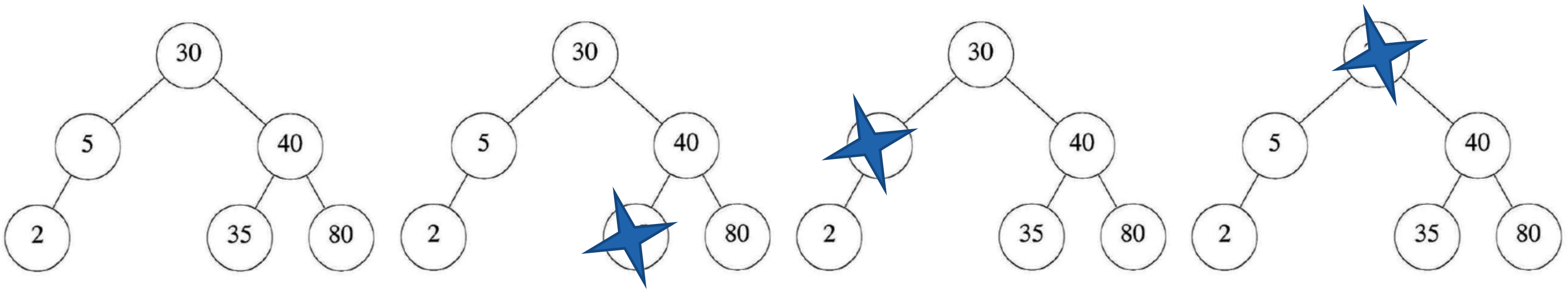
5.7.4 Deletion from a Binary Search Tree

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