Data Structures

Chapter 5 Trees

Additional Binary Tree Operations (1/7)

- **Copying Binary Trees**
  - we can modify the **postorder traversal** algorithm only slightly to copy the binary tree

```
tree_pointer copy(tree_pointer original)
/* this function returns a tree_pointer to an exact copy
of the original tree */
{
   tree_pointer temp;
   if (original) {
      temp = (tree_pointer) malloc(sizeof(node));
      if (IS_FULL(temp)) {
         fprintf(stderr, "The memory is full\n");
         exit(1);
      }
      temp->left_child = copy(original->left_child);
      temp->right_child = copy(original->right_child);
      temp->data = original->data;
      return temp;
   }
   return NULL;
}
```

similar as Program 5.3

Additional Binary Tree Operations (2/7)

- Testing Equality
  - Binary trees are equivalent if they have the same topology and the information in corresponding nodes is identical

V L R

```
int equal(tree_pointer first, tree_pointer 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->left_child,second->left_child) &&
          equal(first->right_child, second->right_child))
}
```

the same topology and data as Program 5.6

Additional Binary Tree Operations (3/7)

- Variables: x_1, x_2, …, x_n can hold only of two possible values, *true* or *false*
- Operators: $\wedge$(***and***), $\vee$(***or***), $\neg$(***not***)
- Propositional Calculus Expression
  - A variable is an expression
  - If x and y are expressions, then $\neg x$, $x \wedge y$, $x \vee y$ are expressions
  - Parentheses can be used to alter the normal order of evaluation ($\neg > \wedge > \vee$)
  - Example: $x_1 \vee (x_2 \wedge \neg x_3)$

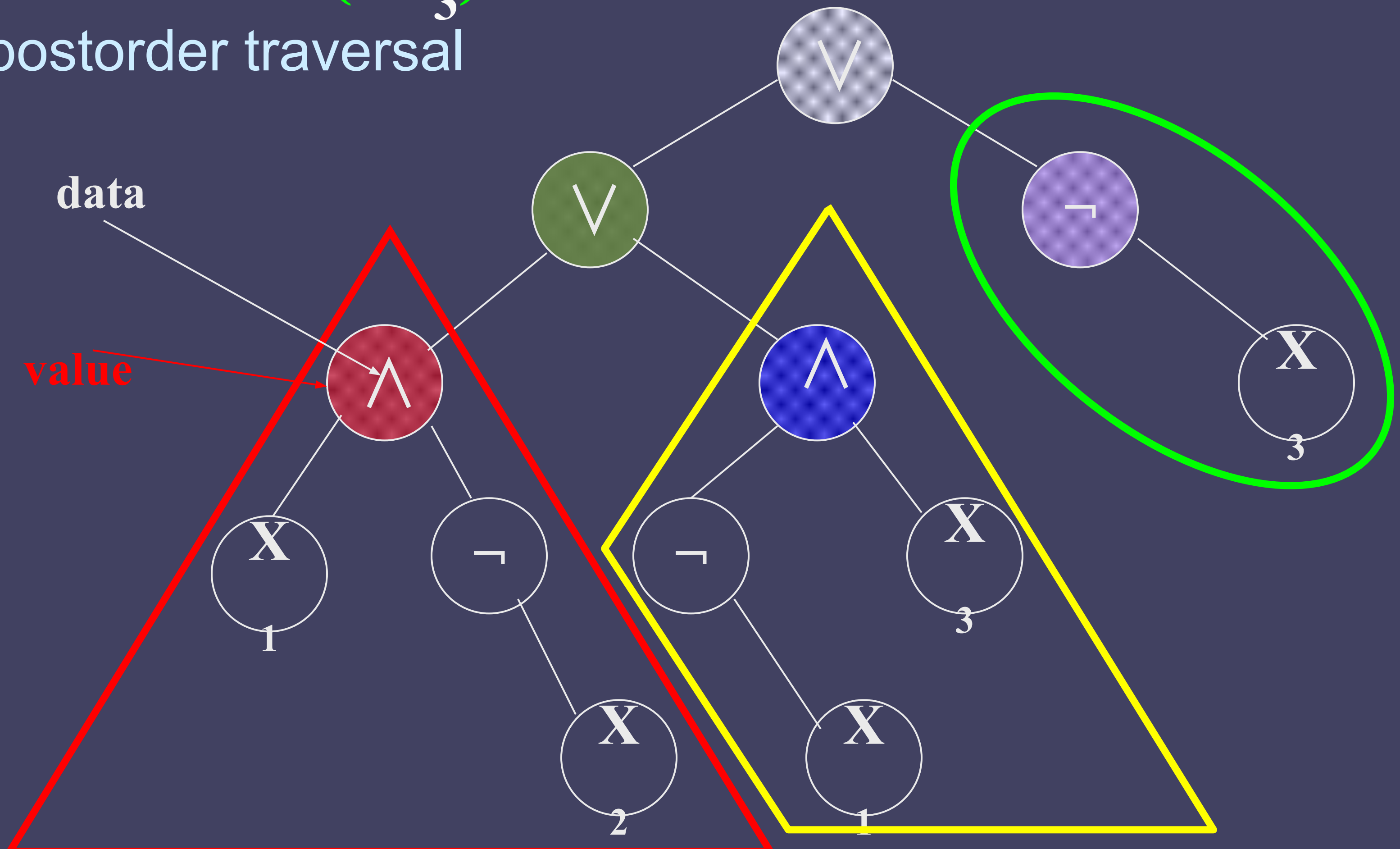
Additional Binary Tree Operations (4/7)

- Satisfiability problem:
  - Is there an assignment to make an expression true?
- Solution for the Example $\boldsymbol{x_1 \vee (x_2 \wedge \neg x_3)}$:
  - If x_1 and x_3 are *false* and x_2 is *true*
  - *false* $\vee$ (*true* $\wedge$ $\neg$*false*) = *false* $\vee$ *true* = *true*
- For n value of an expression, there are 2^n possible combinations of *true* and *false*

Additional Binary Tree Operations (5/7)

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

postorder traversal



Additional Binary Tree Operations (6/7)

- node structure
  - For the purpose of our evaluation algorithm, we assume each node has four fields:

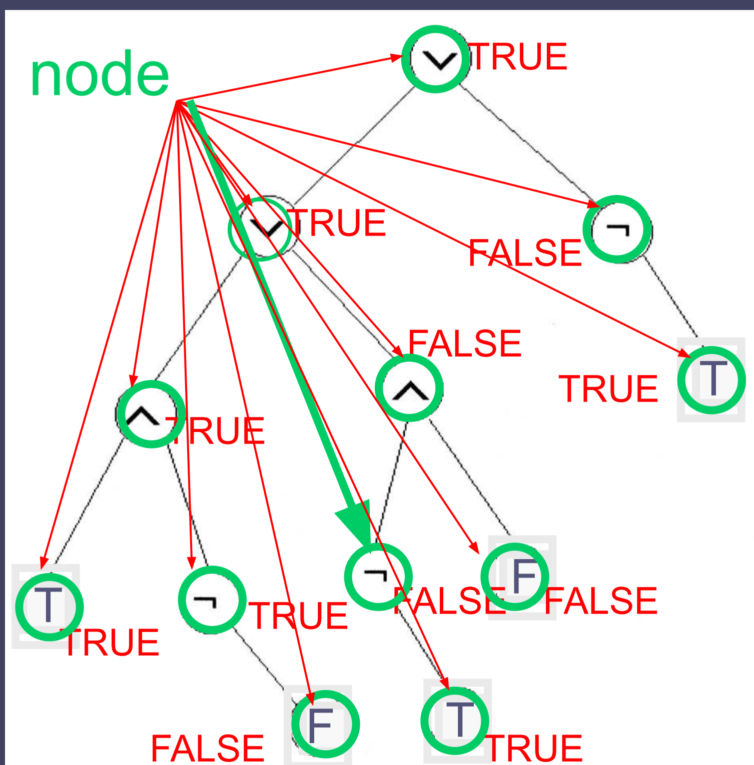
left_child	*data*	*value*	*right_child*

  - We define this node structure in C as:

```
typedef enum {not,and,or,true,false} logical;
typedef struct node *tree_pointer;
typedef struct node {
        tree_pointer left_child;
        logical      data;
        short int    value;
        tree_pointer right_child;
        } ;
```

Additional Binary Tree Operations (7/7)

- **Satisfiability function**
  - To evaluate the tree is easily obtained by modifying the original recursive postorder traversal




```
void post_order_eval(tree_pointer node)
{
/* modified post order traversal to evaluate a
propositional calculus tree */
   if (node) {
      post_order_eval(node->left_child);
      post_order_eval(node->right_child);
      switch(node->data) {
         case not:   node->value =
                 !node->right_child->value;
                 break;
         case and:   node->value =
                 node->right_child->value &&
                 node->left_child->value;
                 break;
         case or:    node->value =
                 node->right_child->value ||
                 node->left_child->value;
                 break;
         case true:  node->value = TRUE;
                 break;
         case false: node->value = FALSE;
      }
   }
}
```

L

R

V

Additional Binary Tree Operations (7/7)

- **Satisfiability function**

```
void post_order_eval(tree_pointer node)
{
/* modified post order traversal to evaluate a
propositional calculus tree */
   if (node) {
      post_order_eval(node->left_child);        L
      post_order_eval(node->right_child);       R
      switch(node->data) {                      V
         case not:    node->value =
                !node->right_child->value;
                break;
         case and:    node->value =
                node->right_child->value &&
                node->left_child->value;
                break;
         case or:     node->value =
                node->right_child->value ||
                node->left_child->value;
                break;
         case true:   node->value = TRUE;
                break;
         case false:  node->value = FALSE;
      }
   }
}
```

Threaded Binary Trees (1/10)

- **Threads**
  - **Too many null pointers** in current representation of binary trees

    ***n*: number of nodes**

    number of non-null links: *n*-1

    total links: 2*n*

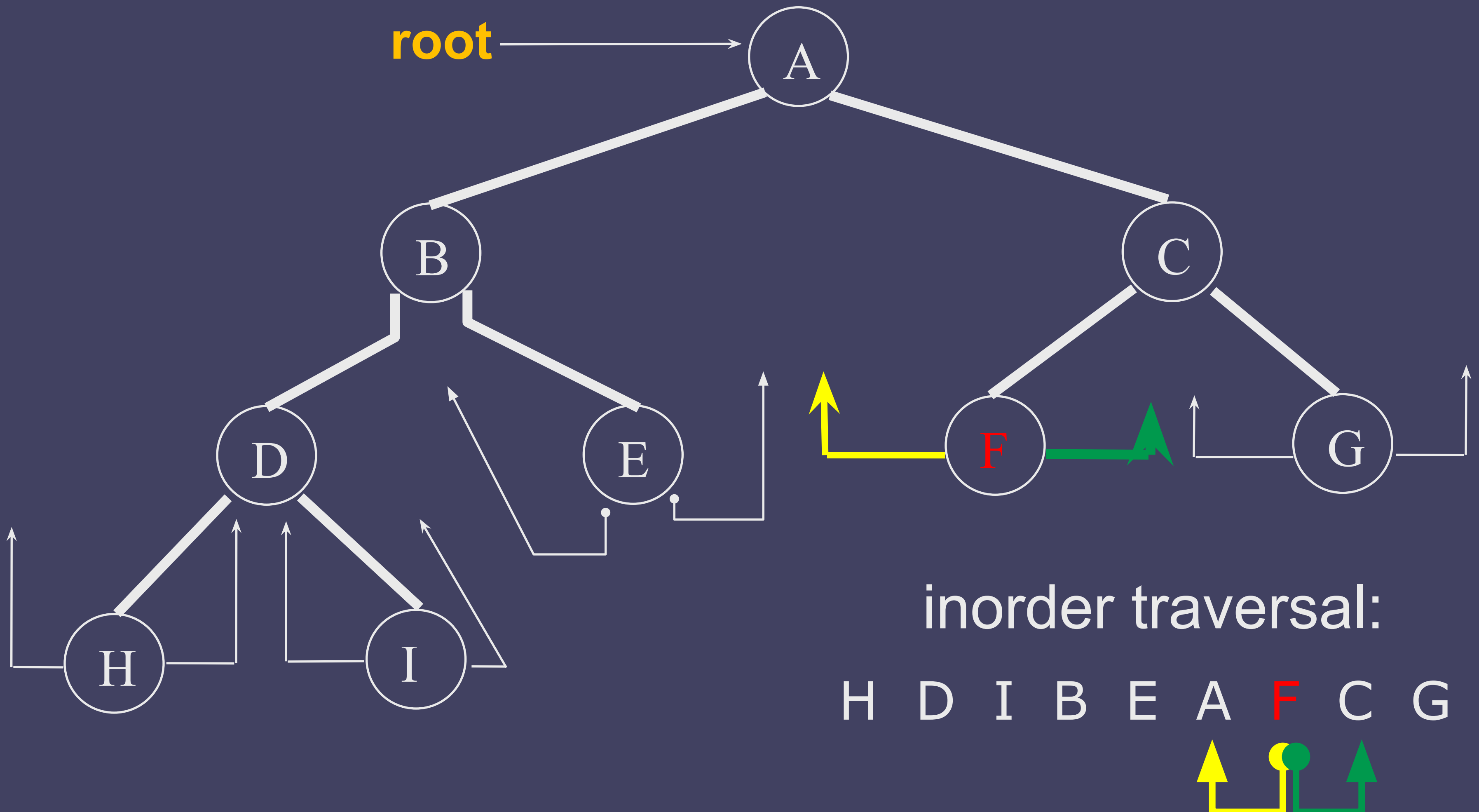
    null links: 2*n*-(*n*-1) = *n*+1
  - Solution: replace these null pointers with some useful "threads"

Threaded Binary Trees (2/10)

- Rules for constructing the threads
  - If **ptr->left_child** is null,
    replace it with a pointer to the node that would be visited **before *`ptr`* *in an inorder traversal***
  - If **ptr->right_child** is null,
    replace it with a pointer to the node that would be visited **after *`ptr`* *in an inorder traversal***

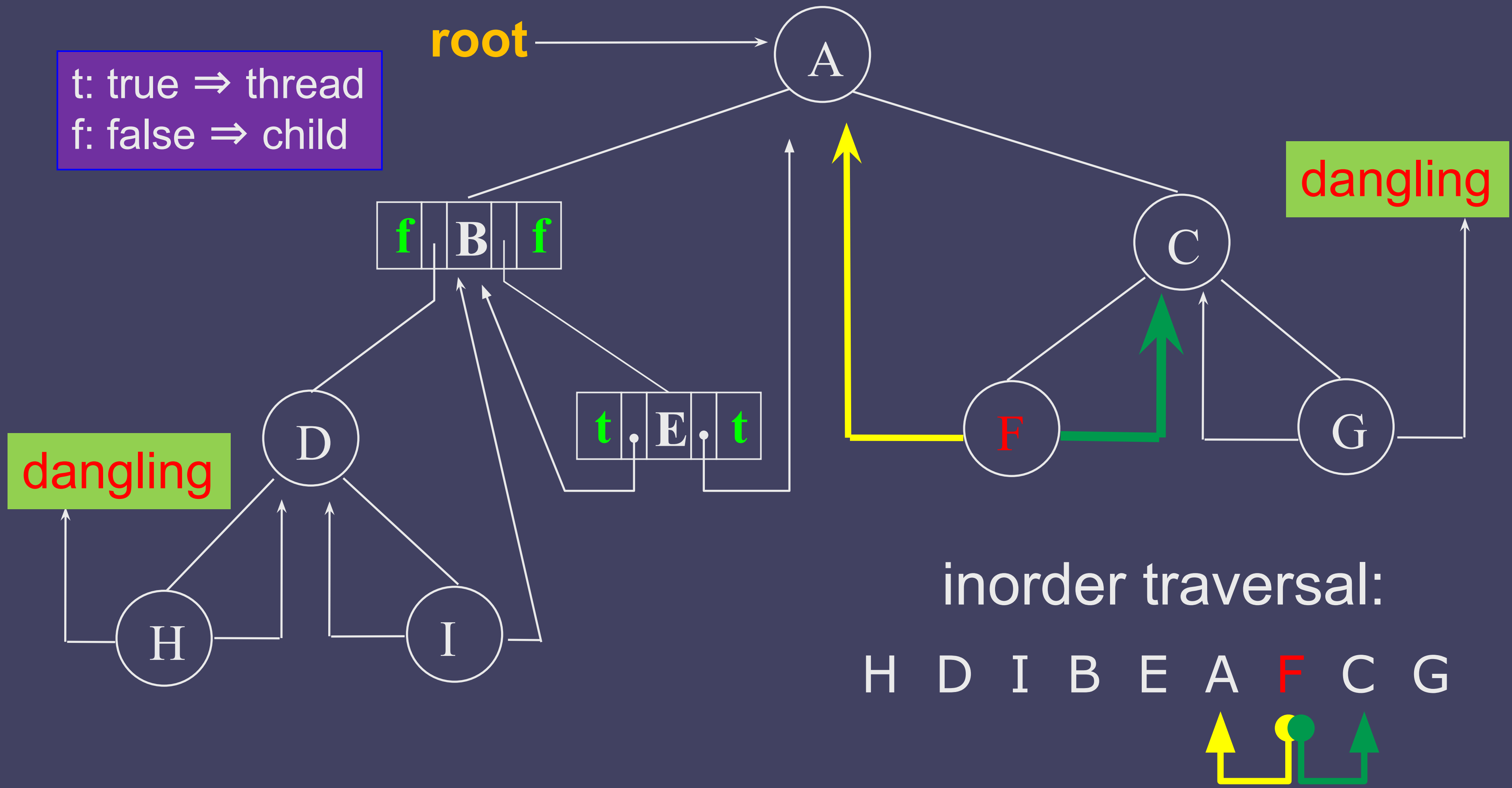
Threaded Binary Trees (3/10)

- **A Binary Tree**



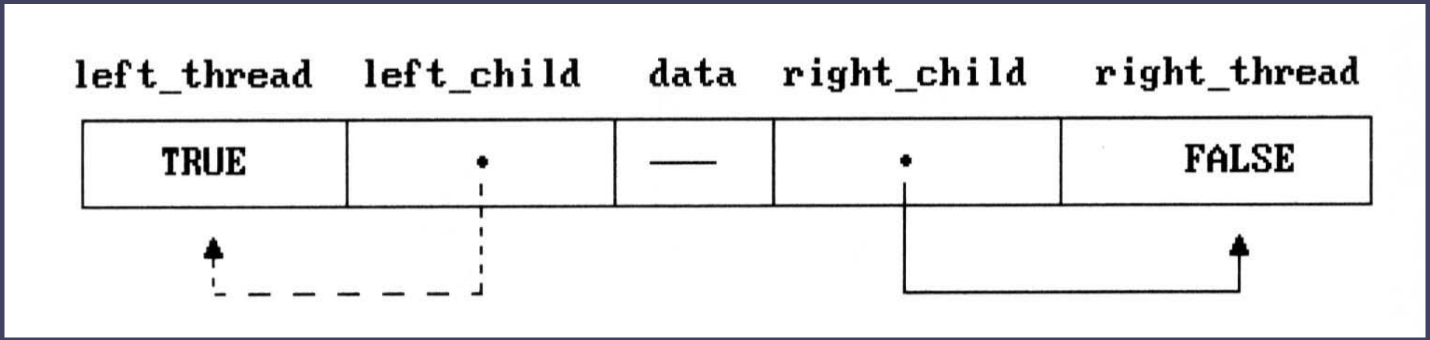
Threaded Binary Trees (3/10)

- **A Threaded Binary Tree**



Threaded Binary Trees (4/10)

- Two additional fields of the node structure, left-thread and right-thread
  - If ptr->left-thread=*TRUE*,
    then ptr->left-child contains a thread;
  - Otherwise it contains a pointer to the left child.
  - Similarly for the right-thread



Threaded Binary Trees (5/10)

- If we don’t want the left pointer of *H* and the right pointer of *G* to be dangling pointers, we may create root node and assign them pointing to the root node

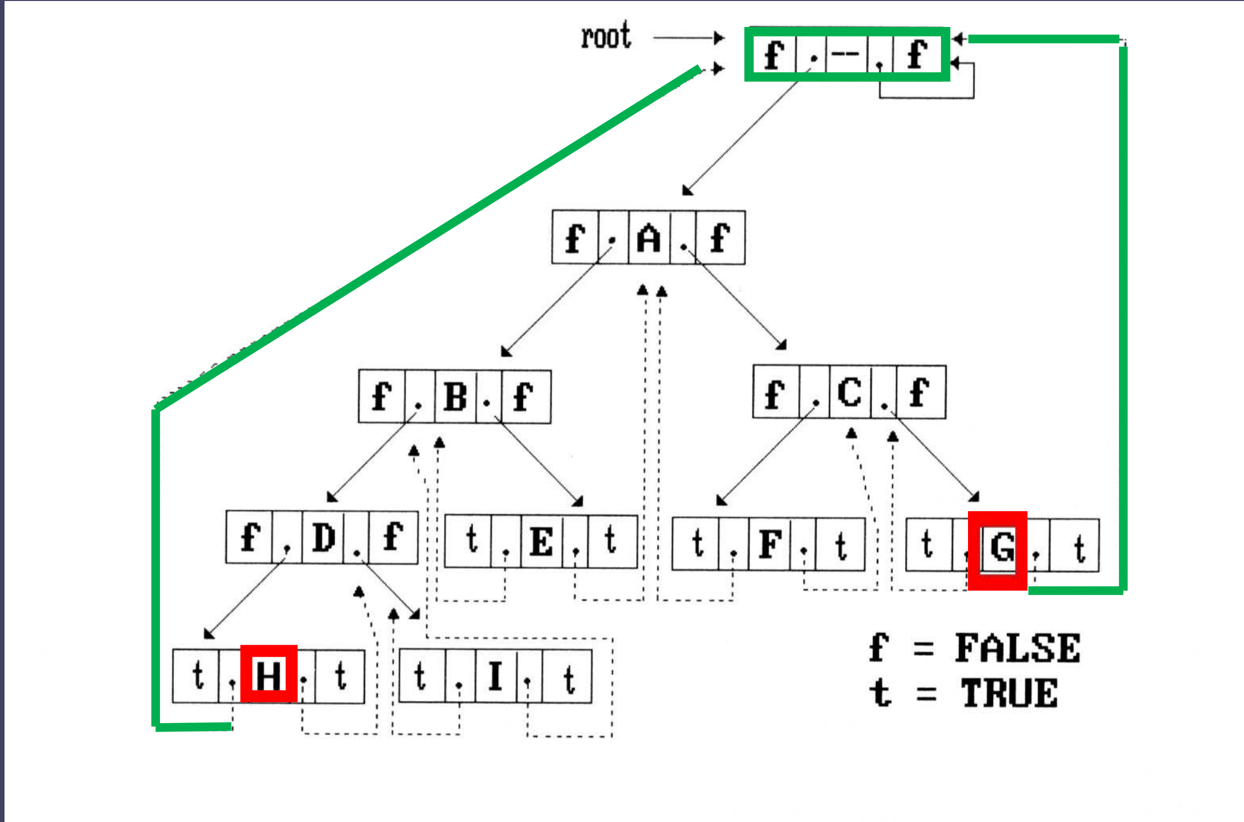


Figure 5.23: Memory representation of a threaded tree

Threaded Binary Trees (6/10)

- **Inorder traversal of a threaded binary tree**
  - By using of threads we can perform an inorder traversal **without making use of a stack** (simplifying the task)

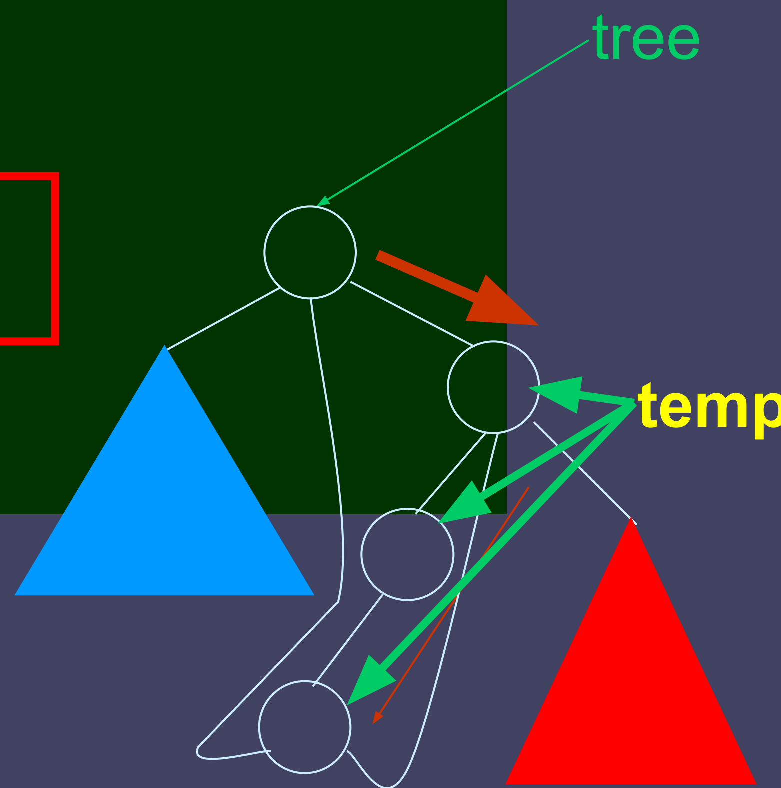
Threaded Binary Trees (6/10)

- Inorder traversal of a threaded binary tree
  1. If *ptr->right_thread = TRUE*, the inorder successor of *ptr* is *ptr->right_child* by definition of the threads
  2. Otherwise we obtain the inorder successor of *ptr* by following a path of left-child links from the right-child of *ptr* until we reach a node with *left_thread = TRUE*

Threaded Binary Trees (7/10)

- Finding the **inorder successor** (next node) of a node

```
threaded_pointer insucc(threaded_pointer tree){
   threaded_pointer temp;
   temp = tree->right_child;
   if (! tree->right_thread)
     while (!temp->left_thread)
          temp = temp->left_child;
   return temp;
}
```

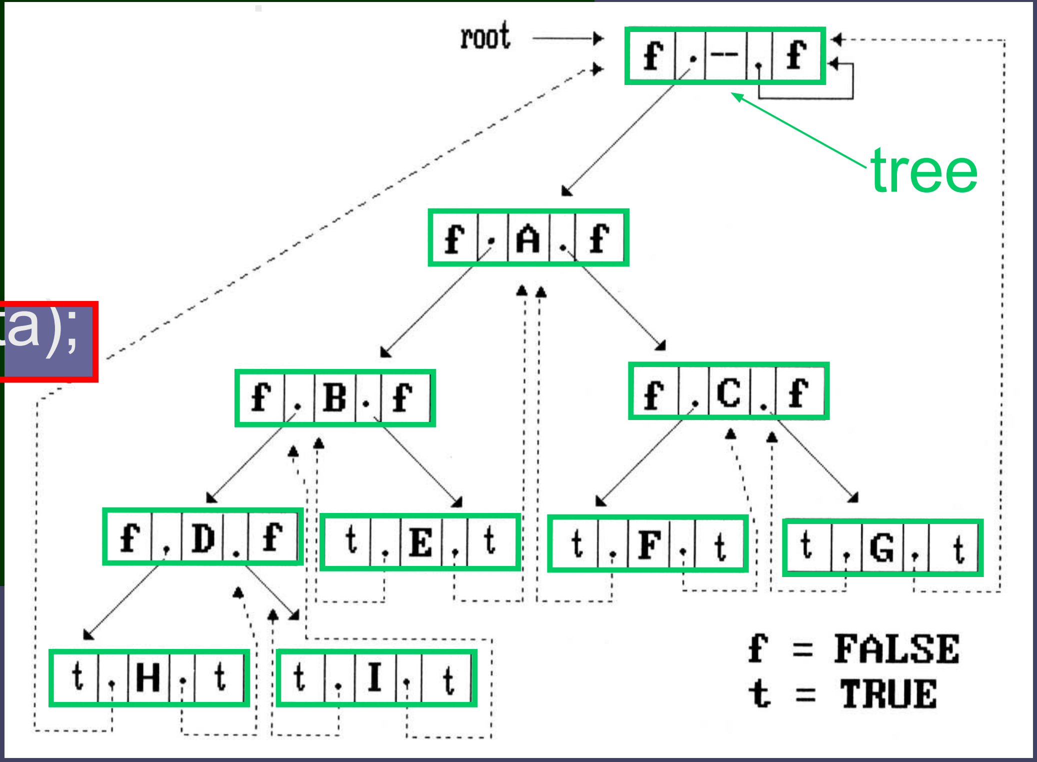


Threaded Binary Trees (8/10)

- Inorder traversal of a threaded binary tree

```
void tinorder(threaded_pointer tree){
/* traverse the threaded binary tree inorder */
   threaded_pointer temp = tree;
   for (;;) {
      temp = insucc(temp);
      if (temp==tree)
            break;
      printf("%3c",temp->data);
   }
}
```

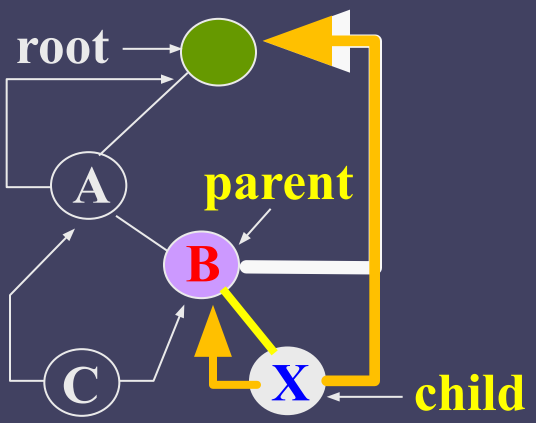
output H D I B E A F C G



Time Complexity: O(n)

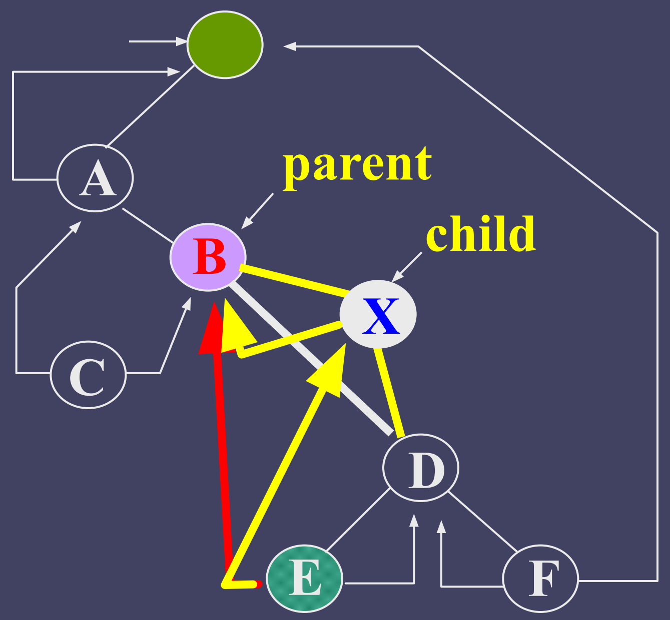
Threaded Binary Trees (9/10)

- Insertion of Threaded Binary Tree
  - Insert *child* as the right child of node *parent*



Threaded Binary Trees (9/10)

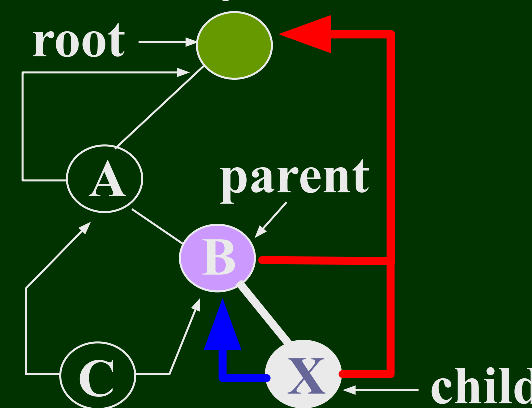
- Insertion of Threaded Binary Tree
  - Insert *child* as the right child of node *parent*

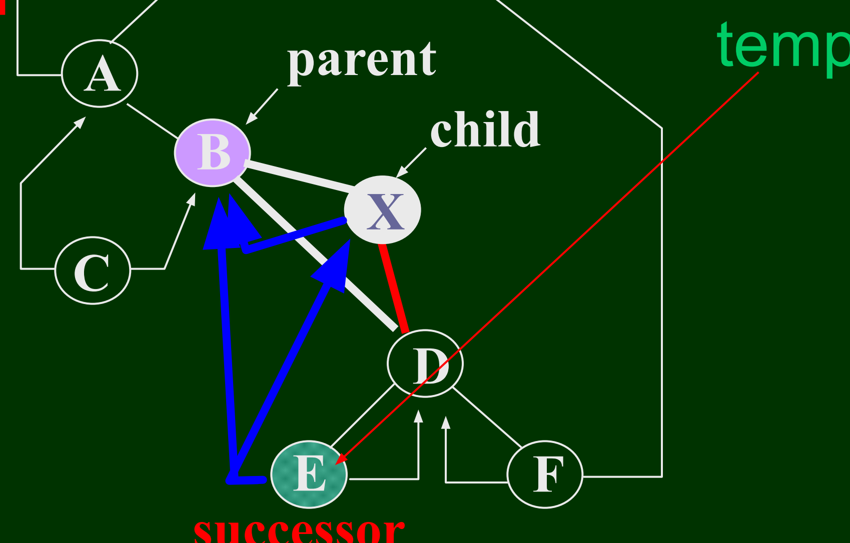


Threaded Binary Trees (10/10)

- Right insertion in a threaded binary tree

```
void insert_right(thread_pointer parent, threaded_pointer child){
/* insert child as the right child of parent in a threaded binary tree */
    threaded_pointer temp;
    child->right_child = parent->right_child;
    child->right_thread = parent->right_thread;
    child->left_child = parent;
    child->left_thread = TRUE;
    parent->right_child = child;
    parent->right_thread = FALSE;
    if(!child->right_thread){
      temp = insucc(child);
      temp->left_child = child;
    }
}
```





First Case

Second Case

Heaps (1/6)

- **Definition:** A *max*(*min*) ***tree*** is a **tree** in which the key value in each node is no smaller (larger) than the key values in its children.

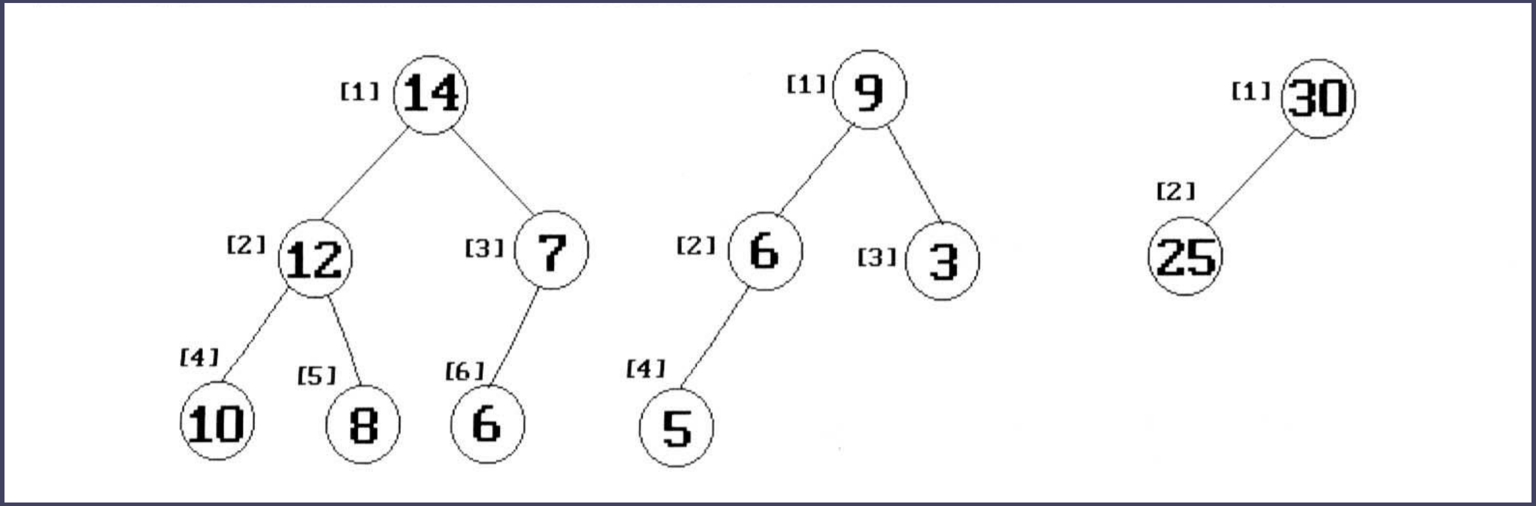
- **Definition:** A *max* (*min*) ***heap*** is a **complete binary** tree that is also a *max* (*min*) *tree*

Heaps (1/6)

- **Basic Operations:**
  - creation of an empty heap
  - insertion of a new element into a heap
  - deletion of the largest element from the heap

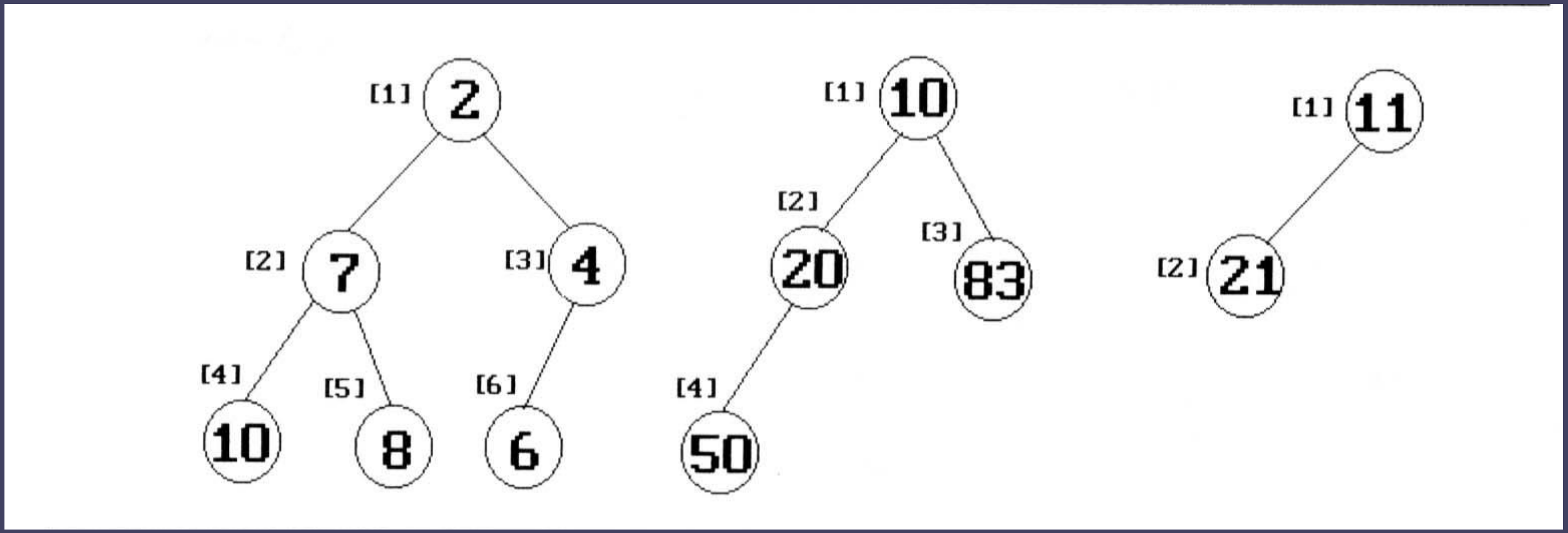
Heaps (2/6)

- The examples of max heaps
  - The root of **max heap** contains the **largest** element



Heaps (2/6)

- The examples of min heaps
- The root of **min heap** contains the **smallest** element



Heaps (4/6)

- **Priority queues**
  - delete the element with highest (lowest) priority
  - insert the element with arbitrary priority

Representation	Insertion	Deletion
Unordered array	$\Theta(1)$	$\Theta(n)$
Unordered linked list	$\Theta(1)$	$\Theta(n)$
Sorted array	$O(n)$	$\Theta(1)$
Sorted linked list	$O(n)$	$\Theta(1)$
Max heap	$O(\log_2 n)$	$O(\log_2 n)$

- **Heaps is the only way to implement priority queue**

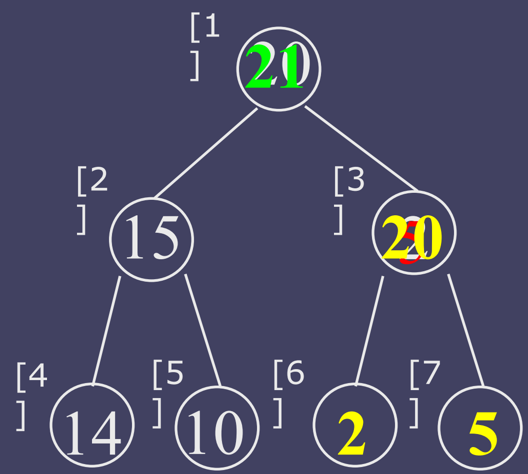
Heaps (5/6)

- **Insertion** Into A **Max Heap**
  - Analysis of *insert_max_heap*
    - The complexity of the insertion function is **O(log$_2$ *n*)**

insert 21

*n=6

i=6



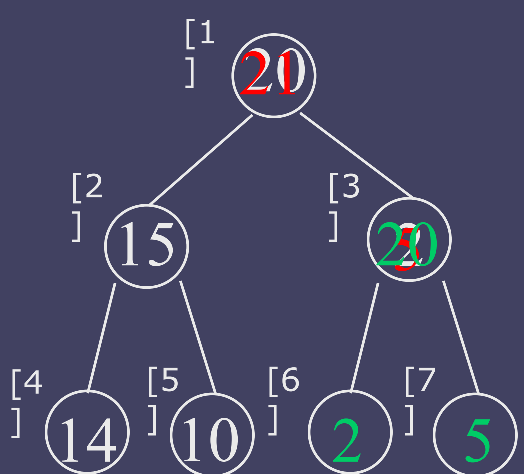
Heaps (5/6)

- **Insertion Into A Max Heap**

insert 21

*n= 6

i= 6




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

parent sink

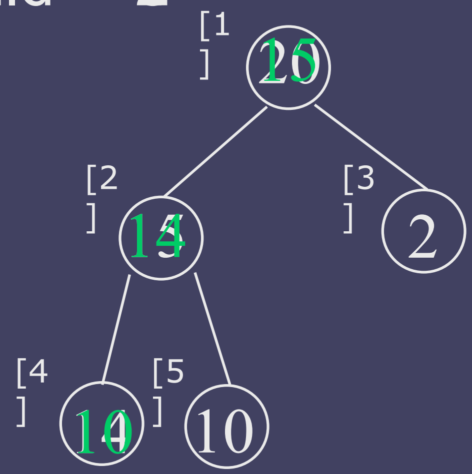
item upheap

Heaps (6/6)

- **Deletion from a max heap**
  - After deletion, the heap is still a complete binary tree
  - Analysis of *delete_max_heap*
    - The complexity of the insertion function is **$O(\log_2 n)$**

parent = 4
child = 8

*n= 5

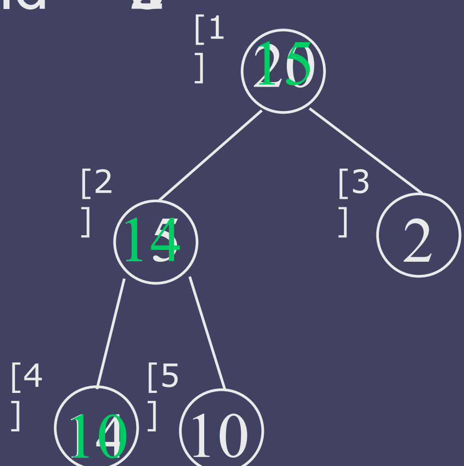


Heaps (6/6)

- **Deletion from a max heap**

parent = 4
child = 8

*n= 4




```
element delete_max_heap(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(1);
   }
   /* 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;
}
```

item.key = 20
temp.key = 10

Binary Search Trees (1/8)

- Why do binary search trees need?
  - Heap is not suited for applications in which arbitrary elements are to be deleted from the element list
    - a min (max) element is deleted $O(\log_2 n)$
    - deletion of an arbitrary element $O(n)$
    - search for an arbitrary element $O(n)$

Binary Search Trees (1/8)

- **Definition** of binary search tree:
  - Every element has a unique key
  - The keys in a nonempty left subtree (right subtree) are smaller (larger) than the key in the root of subtree
  - The left and right subtrees are also binary search trees

Binary Search Trees (2/8)

- Example: (b) and (c) are binary search trees

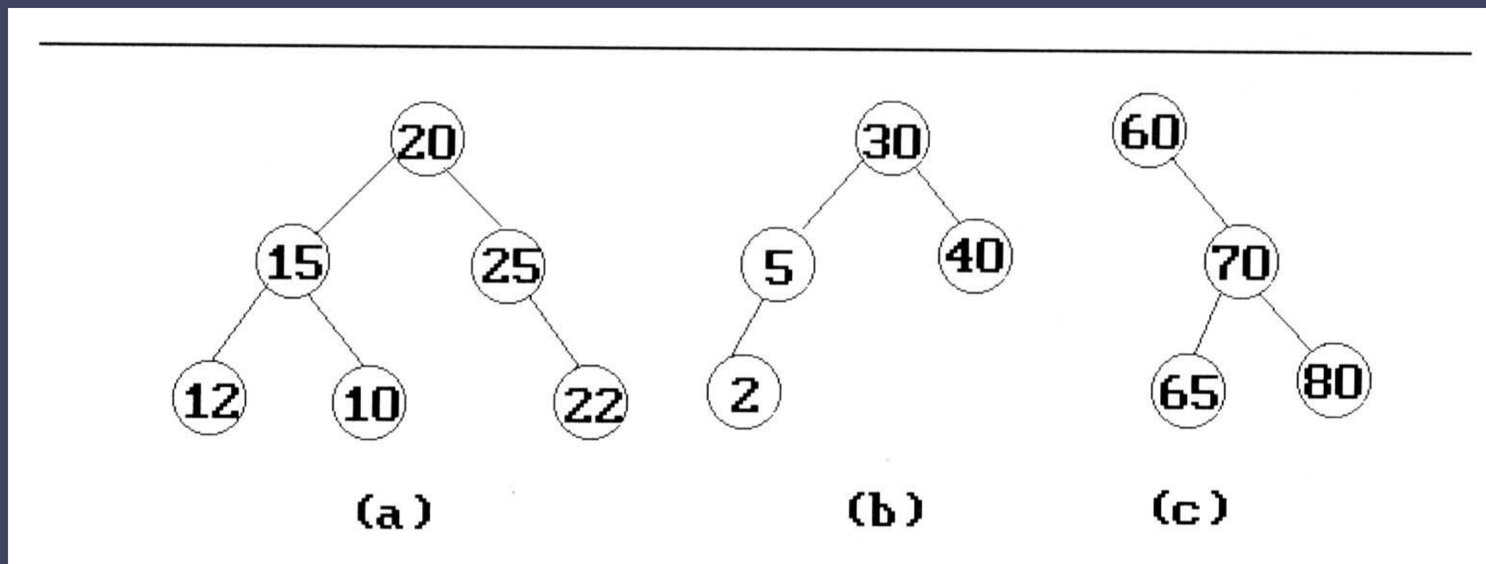
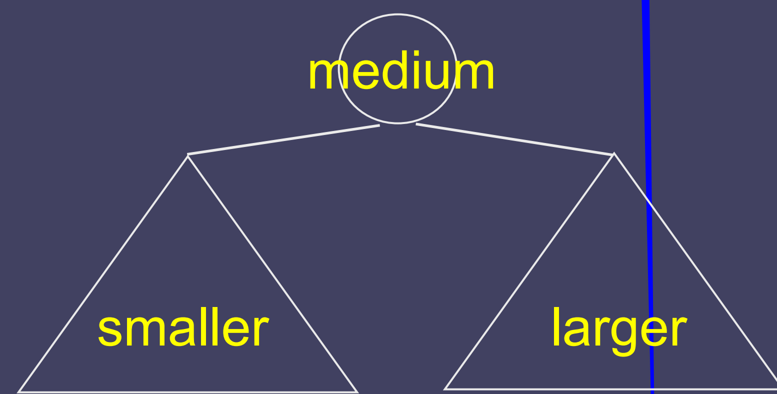
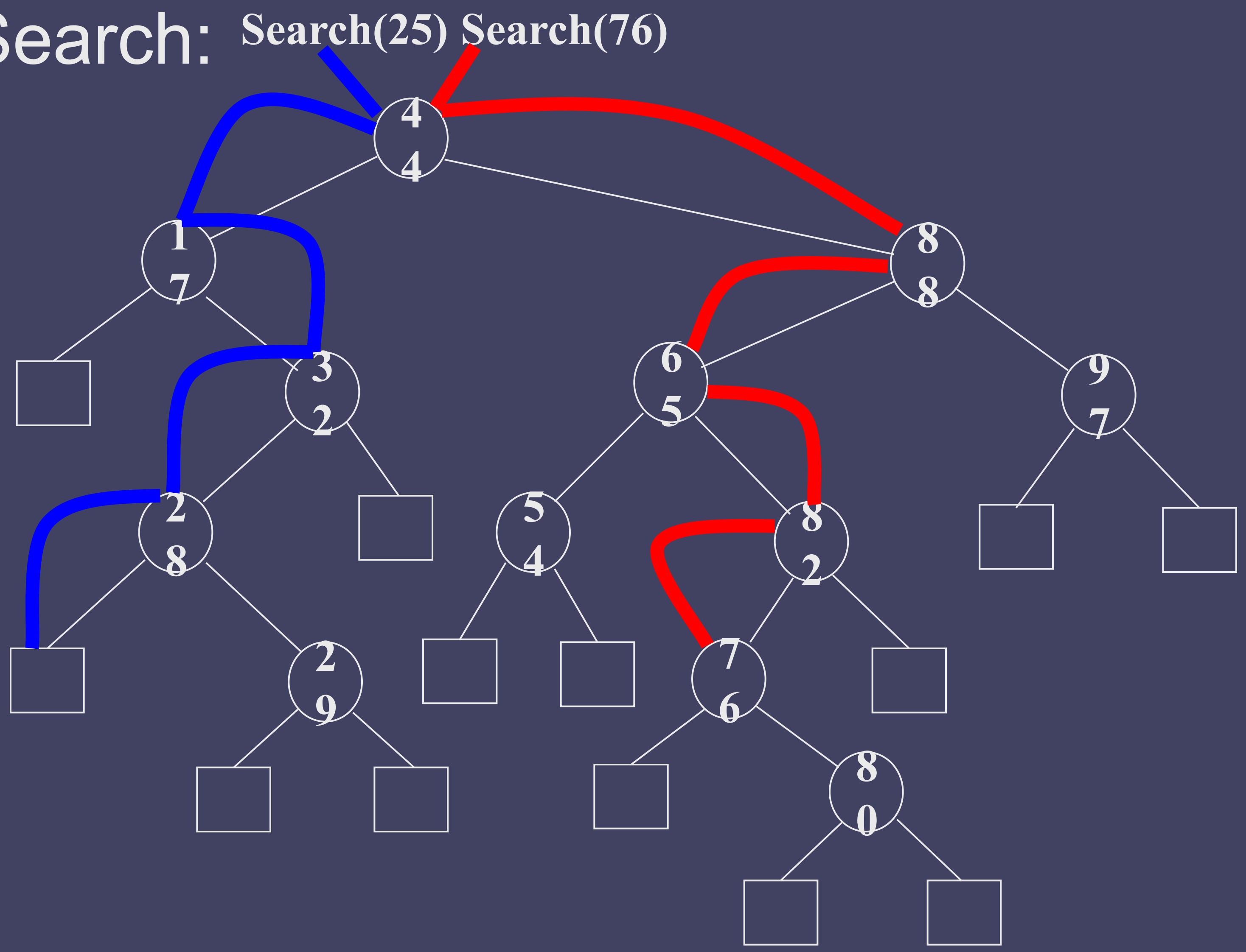


Figure 5.30: Binary trees



Binary Search Trees (3/8)

- Search:



Binary Search Trees (4/8)

- Searching a binary search tree

```
tree_pointer search(tree_pointer root, int key)
{
/* return a pointer to the node that contains key.  If
there is no such node, return NULL. */
   if (!root) return NULL;
   if (key == root->data) return root;
   if (key < root->data)
      return search(root->left_child, key);
   return search(root->right_child,key);
}
```

Program 5.15: Recursive search of a binary search tree

Binary Search Trees (4/8)

- Searching a binary search tree

O(h)

```
tree_pointer search2(tree_pointer tree, int key)
{
/* return a pointer to the node that contains key.  If
there is no such node, return NULL. */
   while (tree) {
      if (key == tree->data) return tree;
      if (key < tree->data)
         tree = tree->left_child;
      else
         tree = tree->right_child;
      }
   return NULL;
}
```

Program 5.16: Iterative search of a binary search tree

Binary Search Trees (5/8)

- Inserting into a binary search tree

```
void insert_node(tree_pointer *node, int num)
/* If num is in the tree pointed at by node do nothing;
otherwise add a new node with data = num */
{
   tree_pointer ptr, temp = modified_search(*node, num);
   if (temp || !(*node)) {
      /* num is not in the tree */
      ptr = (tree_pointer)malloc(sizeof(node));
      if (IS_FULL(ptr)) {
         fprintf(stderr, "The memory is full\n");
         exit(1);
      }
      ptr->data = num;
      ptr->left_child = ptr->right_child = NULL;
      if (*node) /* insert as child of temp */
         if (num < temp->data) temp->left_child = ptr;
         else temp->right_child = ptr;
      else *node = ptr;
   }
}
```

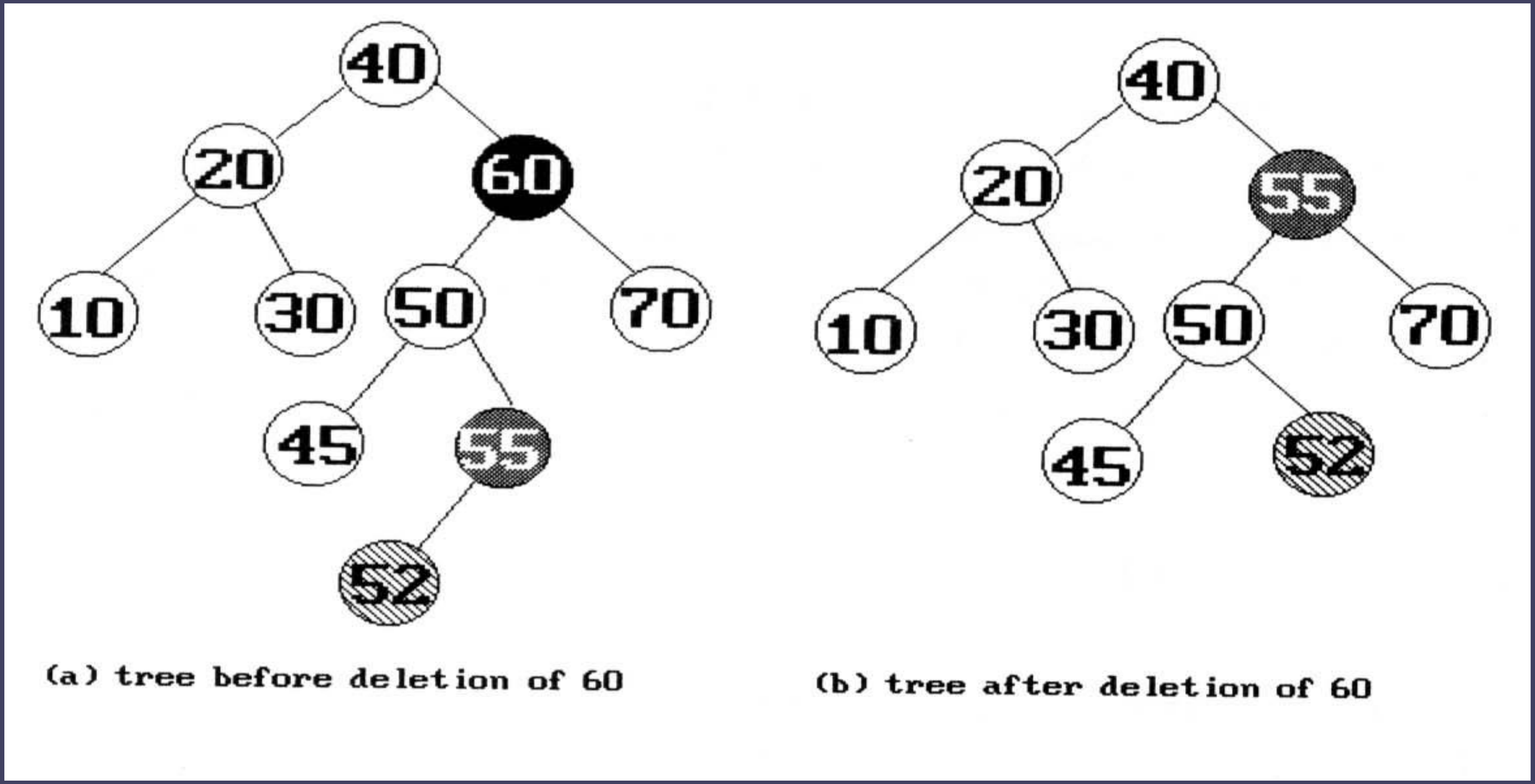
An empty tree

參下方備忘稿

Program 5.17: Inserting an element into a binary search tree

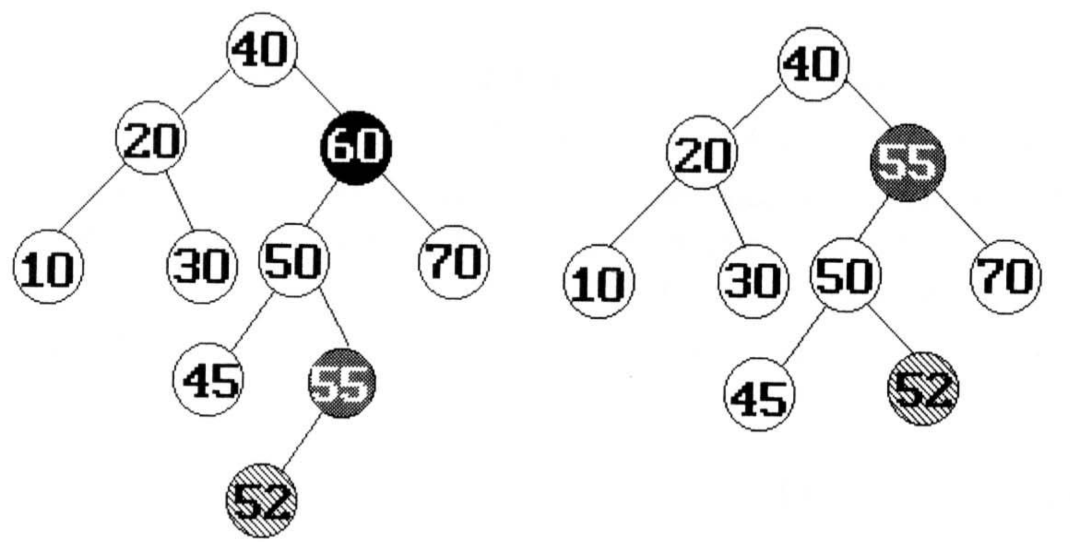
Binary Search Trees (6/8)

- **Deletion from a binary search tree**



Binary Search Trees (6/8)

- **Deletion from a binary search tree**
  - Three cases should be considered
  - case 1. leaf → delete
  - case 2. one child → delete and change the pointer to this child
    case 3. two child → either the smallest element in the right subtree or the largest element in the left subtree



(a) tree before deletion of 60 (b) tree after deletion of 60

Binary Search Trees (7/8)

- Height of a binary search tree
  - The height of a binary search tree with n elements can become as large as n.
  - It can be shown that when insertions and deletions are made at random, the height of the binary search tree is $O(\log_2 n)$ on the average.
  - Search trees with a worst-case height of $O(\log_2 n)$ are called ***balance search trees***

Binary Search Trees (8/8)

- Time Complexity
  - Searching, insertion, removal
    - **O(*h*)**, where h is the height of the tree
  - Worst case - skewed binary tree
    - **O(*n*)**, where n is the # of internal nodes
- Prevent worst case
  - **rebalancing scheme**
  - AVL, 2-3, and Red-black tree

Selection Trees (1/7)

- Problem:
  - suppose we have *k* order sequences, called runs, that are to be merged into a single ordered sequence
- Solution: (for determine maximum/minimum)
  - straightforward : k-1 comparison
  - selection tree : $\lceil \log_2 k \rceil + 1$
- There are two kinds of selection trees: *winner trees* and *loser trees*

Selection Trees (2/7)

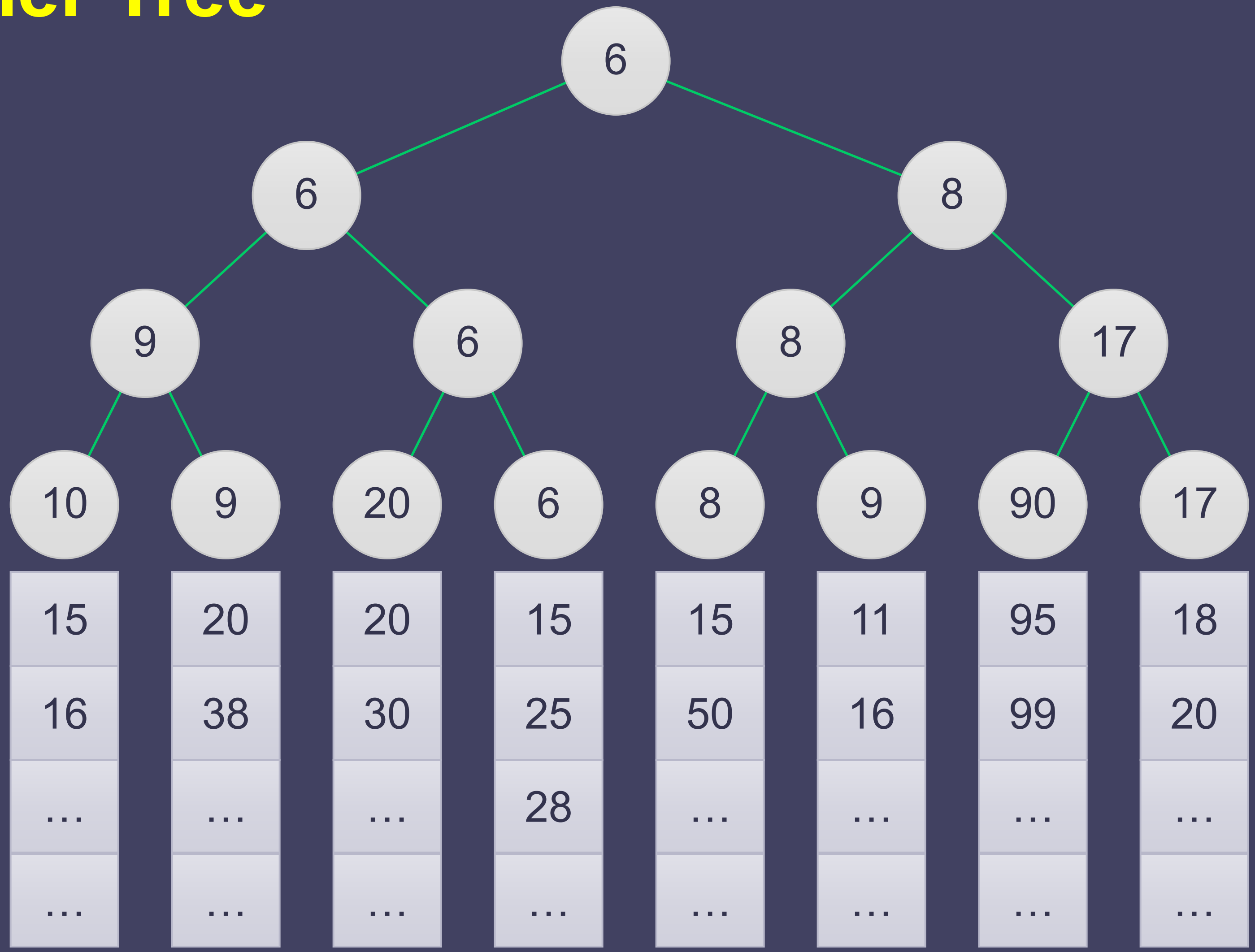
- Definition: (Winner tree)
  - a selection tree is the binary tree where each node represents the smaller of its two children
  - root node is the smallest node in the tree
  - a winner is the record with smaller key
- Rules:
  - tournament : between sibling nodes
  - put X in the parent node ⇒ X win

- Input: k ordered sequences

10	9	20	6	8	9	90	17
15	20	20	15	15	11	95	18
16	38	30	25	50	16	99	20
…	…	…	28	…	…	…	…
…	…	…	…	…	…	…	…

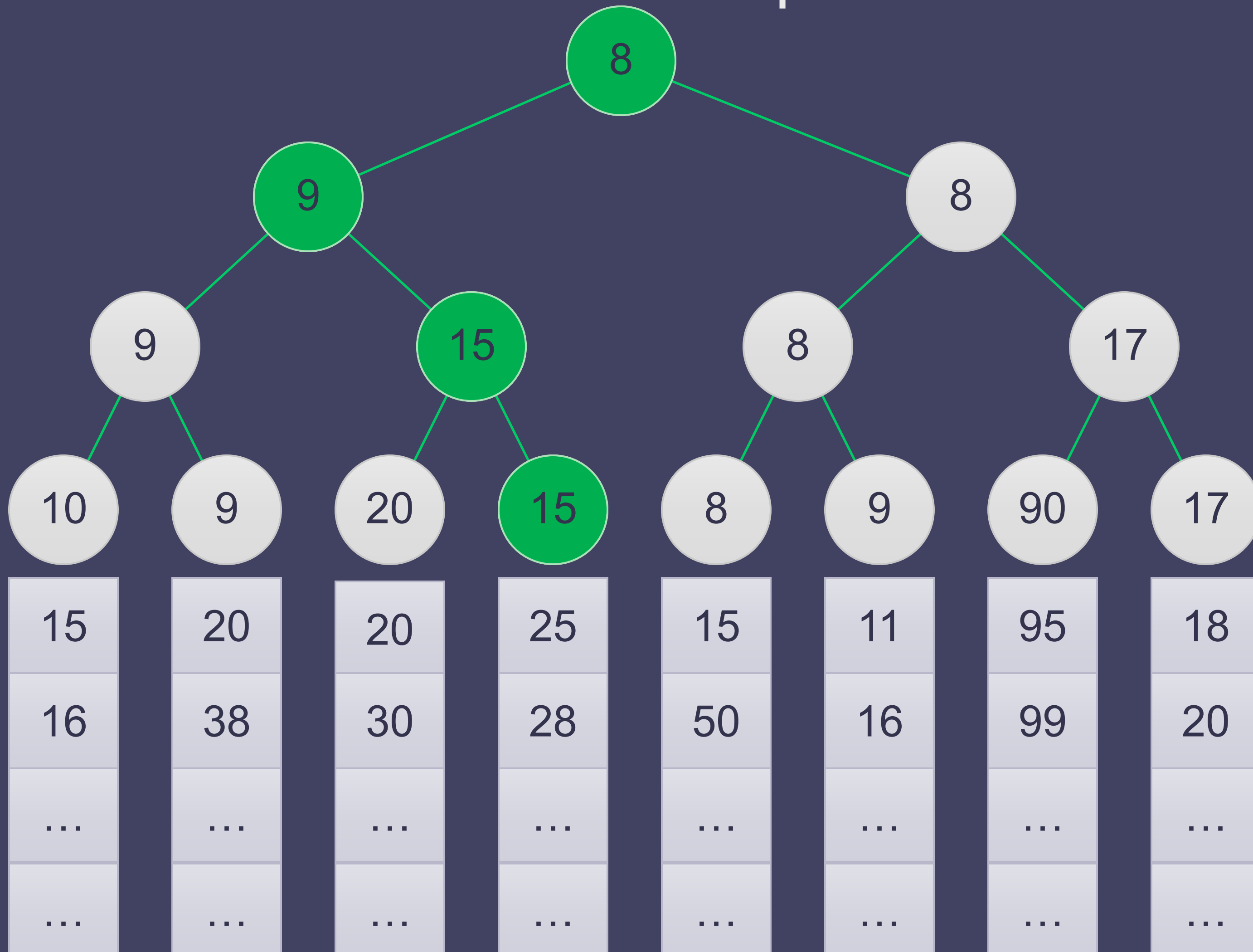
Selection Trees (3/7)

- **Winner Tree**



Selection Trees (5/7)

- After one record has been output



Selection Trees (6/7)

- Tree of losers can be conducted by Winner tree

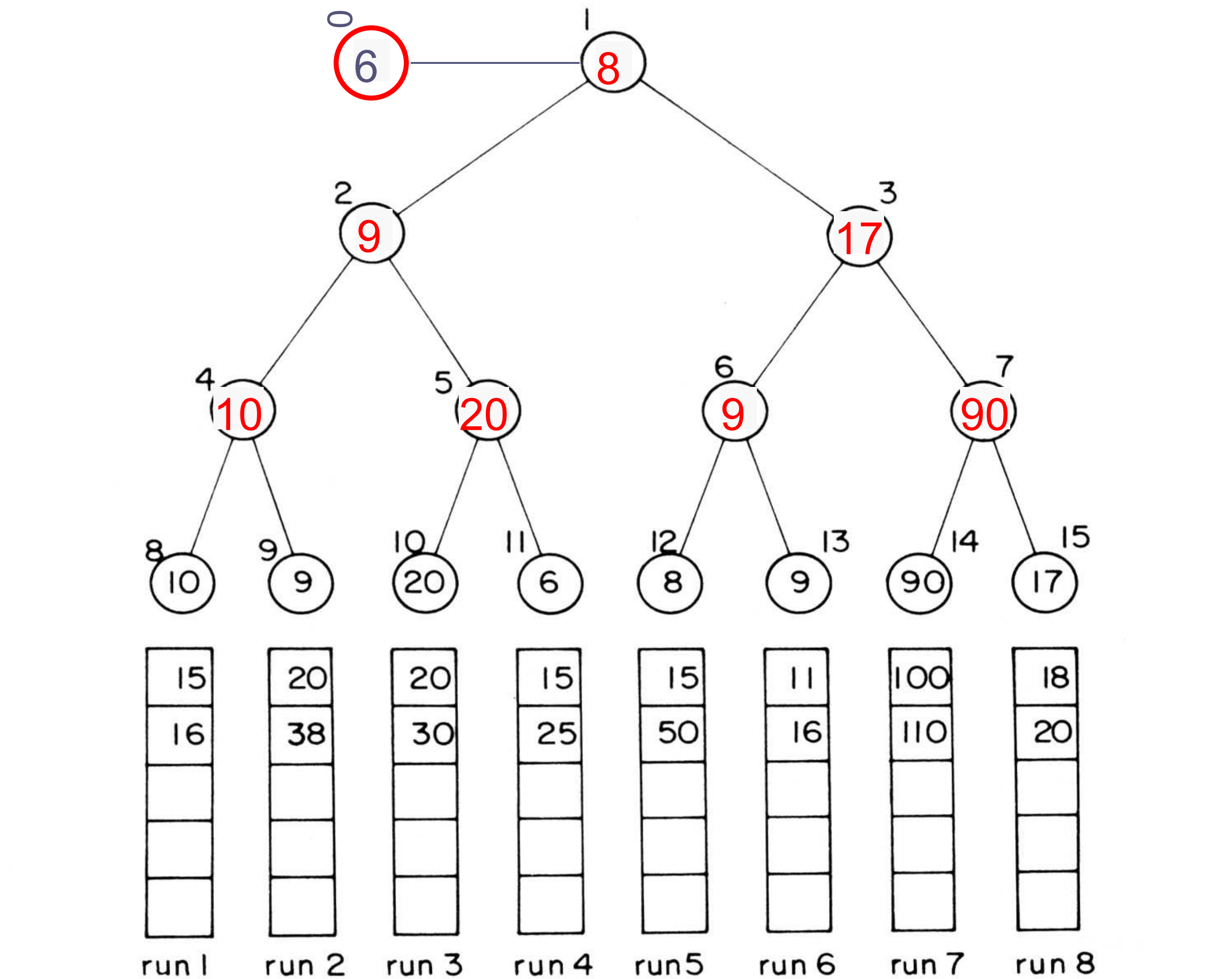


Figure 5.34: Selection tree for $k = 8$ showing the first three keys in each of the eight runs

Selection Trees (4/7)

- Analysis of merging runs using winner trees
  - # of levels: $\lceil \log_2 K \rceil + 1 \Rightarrow$ restructure time: $O(\log_2 K)$
  - merge time: $O(n\log_2 K)$
  - setup time: O(K)
  - merge time: $O(n\log_2 K)$
- Slight modification: **tree of loser**
  - consider the parent node only (vs. sibling nodes)

Selection Trees (7/7)

- The loser tree after output 6