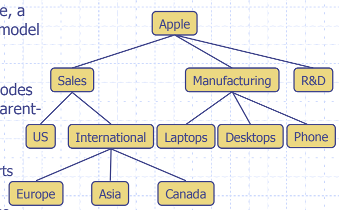
6. Trees, Heaps, and Binary Search Trees (BSTs)

- Trees: tree terminology, tree ADT, operations and efficiency
- Binary trees
- Heaps: heap applications, heap operations and efficiency, heap as priority queue, heap sort
- BSTs: operations and efficiency

Trees

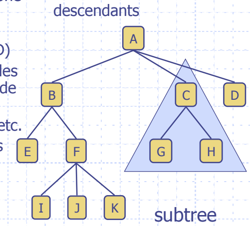
What is a Tree

- In computer science, a tree is an abstract model of a hierarchical structure
- A tree consists of nodes and edges with a parent-child relation
- Applications:
  - Organization charts
  - File systems
  - Decision processes



Tree Terminology

- Root: node without parent (A)
- Internal node: node with at least one child (A, B, C, F)
- External node (a.k.a. leaf): node without children (E, I, J, K, G, H, D)
- Path to a node: a sequence of nodes and edges from the root to the node
- Ancestors of a node: parent, grandparent, grand-grandparent, etc.
- Depth of a node: number of edges on the path to the node
- Height of a tree: the number of edges on the longest path
- Descendant of a node: child, grandchild, grand-grandchild, etc.
- Subtree: tree consisting of a node and its descendants



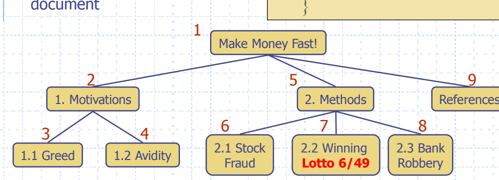
Tree ADT

- Generic functions:
  - int size()
  - boolean isEmpty()
- Access functions:
  - node root(tree)
  - node parent(p)
  - node_list children(p)
  - node left_child(p)
  - node right_child(p)
- Query functions:
  - boolean isInternal(p)
  - boolean isExternal(p)
  - boolean isRoot(p)
  - int height(tree)
- Additional update functions may be defined by data structures implementing the Tree ADT

Preorder Traversal

- A traversal visits the nodes of a tree in a systematic manner
- In a preorder traversal, a node is visited before its descendants
- Application: print a structured document

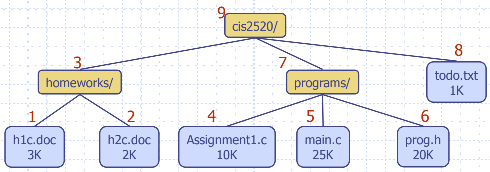
```
function preOrder(v)
    visit(v);
    for (each w of children(v)){
        preOrder (w);
    }
```



Postorder Traversal

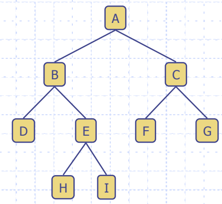
- In a postorder traversal, a node is visited after all its descendants
- Application: compute space used by files in a directory and its subdirectories

```
function postOrder(v)
    for (each w of children(v)){
        postOrder (w);
    }
    visit(v);
```



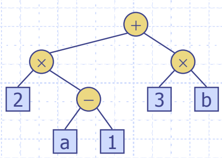
Binary Tree

- A binary tree is a tree in which each internal node has at most two children
- We call the children of an internal node left child and right child
- Alternative recursive definition: a binary tree is either
  - a tree consisting of a single node, or
  - a tree whose root has one or two children, each of which is a binary tree
- Applications:
  - arithmetic expressions
  - decision processes
  - searching



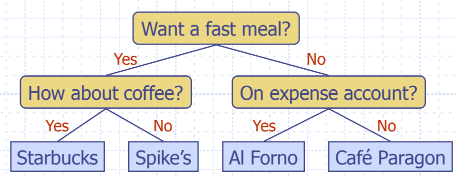
Arithmetic Expression Tree

- Binary tree associated with an arithmetic expression
  - internal nodes: operators
  - external nodes: operands
- Example: arithmetic expression tree for the expression $(2 \times (a - 1) + (3 \times b))$



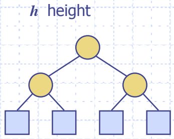
Decision Tree

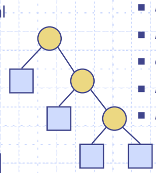
- Binary tree associated with a decision process
  - internal nodes: questions with yes/no answer
  - external nodes: decisions
- Example: dining decision



Properties of Binary Trees

- Notation
  - n number of nodes
  - e number of external nodes
  - i number of internal nodes
  - h height



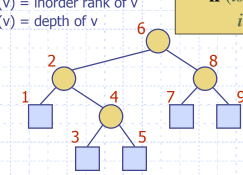


- Properties:
  - $e = i + 1$
  - $n = 2e - 1$
  - $h \leq i$
  - $h \leq (n - 1)/2$
  - $e \leq 2^h$
  - $h \geq \log_2 e$
  - $h \geq \log_2 (n + 1) - 1$

Inorder Traversal

- In an inorder traversal a node is visited after its left subtree and before its right subtree
- Application: draw a binary tree
  - x(v) = inorder rank of v
  - y(v) = depth of v

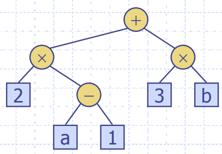
```
function inOrder(v)
    if (isInternal (v)){
        inOrder (leftChild (v));}
    visit(v);
    if (isInternal (v)){
        inOrder (rightChild (v));}
```



Print Arithmetic Expressions

- Specialization of an inorder traversal
  - print "(" before traversing left subtree
  - print operand or operator when visiting node
  - print ")" after traversing right subtree

```
function printExpression(v)
    if (isInternal (v)){
        printf("%c", `(');
        inOrder (leftChild (v));
    }
    print(v); // depends on v's data
    if (isInternal (v)){
        inOrder (rightChild (v));
        printf("%c", `)');
    }
```

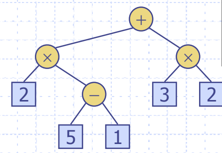


$((2 \times (a - 1)) + (3 \times b))$

Evaluate Arithmetic Expressions

- Specialization of a postorder traversal
  - recursive method returning the value of a subtree
  - when visiting an internal node, combine the values of the subtrees

```
function evalExpr(v)
    if (isExternal (v)){
        return v.value ();
    }
    else{
        x = evalExpr(leftChild (v));
        y = evalExpr(rightChild (v));
        ◊ ← operator stored at v;
        return x ◊ y;
    }
```



Priority Queue ADT

- A priority queue stores a collection of items
- An item is a pair (key, element)
- Main methods of the Priority Queue ADT
  - insertItem(k, o) inserts an item with key k and element o
  - removeMin() removes the item with smallest key and returns its element
- Additional methods
  - minKey() returns, but does not remove, the smallest key of an item
  - minElement() returns, but does not remove, the element of an item with smallest key
  - size(), isEmpty()
- Applications:
  - Standby flyers
  - Auctions

Example: Priority Queue

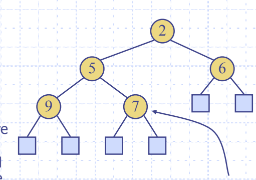
Operator	Output	Priority Queue
insertItem(5, A)	_	(5,A)
insertItem(9, C)	_	(5,A),(9,C)
insertItem(3, B)	_	(3,B),(5,A),(9,C)
insertItem(7, D)	_	(3,B),(5,A),(7,D),(9,C)
minElement()	B	(3,B),(5,A),(7,D),(9,C)
minKey()	3	(3,B),(5,A),(7,D),(9,C)
removeMin()	B	(5,A),(7,D),(9,C)
size()	3	(5,A),(7,D),(9,C)
removeMin()	A	(7,D),(9,C)
removeMin()	D	(9,C)
removeMin()	C	
removeMin()	error	
isEmpty()	true	

Total Order Relation

- Keys in a priority queue can be arbitrary objects on which an order is defined
- Two distinct items in a priority queue can have the same key
- Mathematical concept of total order relation $\leq$
  - Reflexive property: $x \leq x$
  - Antisymmetric property: $x \leq y \wedge y \leq x \Rightarrow x = y$
  - Transitive property: $x \leq y \wedge y \leq z \Rightarrow x \leq z$

What is a heap

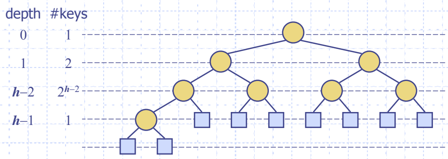
- A heap is a binary tree storing keys at its internal nodes and satisfying the following properties:
  - Heap-Order: for every internal node v other than the root, $key(v) \geq key(parent(v))$
  - Complete Binary Tree: let h be the height of the heap
    - for $i = 0, \ldots, h - 2$, there are 2^i nodes of depth i
    - at depth $h - 1$, the internal nodes are to the left of the external nodes
- The last node of a heap is the rightmost internal node of depth $h - 1$



Height of a Heap

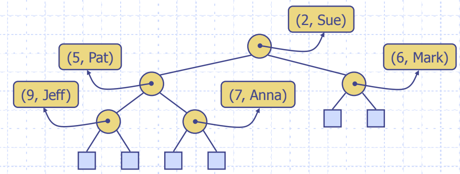
- Theorem: A heap storing n keys has height $O(\log n)$

  Proof: (we apply the complete binary tree property)
  - Let h be the height of a heap storing n keys
  - Since there are 2^i keys at depth $i = 0, \ldots, h-2$ and at least one key at depth $h-1$, we have $n \geq 1 + 2 + 4 + \ldots + 2^{h-2} + 1$
  - Thus, $n \geq 2^{h-1}$, i.e., $h \leq \log n + 1$



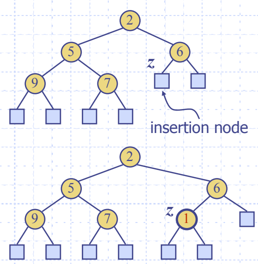
Heaps and Priority Queues

- We can use a heap to implement a priority queue
- We store a (key, element) item at each internal node
- We keep track of the position of the last node
- For simplicity, we show only the keys in the pictures



Insertion into a Heap

- The insertion algorithm consists of three steps
  - Find the insertion position z (always the new last node)
  - Store k at z and expand z into an internal node
  - Restore the heap-order property (discussed next)



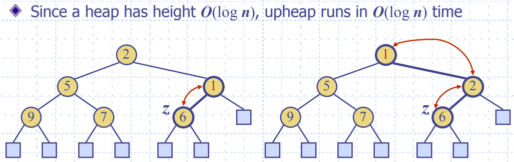

```
void insert(Heap *H, ItemType ItemToInsert) {
NodeType  *N;                    /* Pointer to the new node to be inserted to H */
NodeType  *P;                    /* Let P be the pointer to the new node */
        N = (create a new node with ItemToInsert);
        if  (*H is not empty)  {
            /* Find the position to hold the new node */
            P = (the pointer to the new node);
            (P's value) = N;
            /* Re-heapify the values in the remaining nodes of H starting at the root, R */

               (Re-heapify the heap H starting at node R);

        }
        else {
            (Root in H) = N;
        }
        return;
}
```

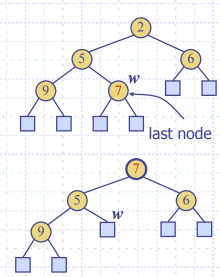

Upheap

- After the insertion of a new key k, the heap-order property may be violated
- Algorithm upheap restores the heap-order property by swapping k along an upward path from the insertion node
- Upheap terminates when the key k reaches the root or a node whose parent has a key smaller than or equal to k
- Since a heap has height $O(\log n)$, upheap runs in $O(\log n)$ time



Removal from a Heap

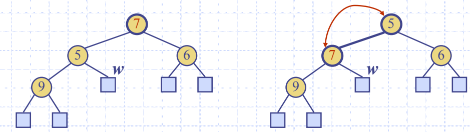
- The removal algorithm consists of three steps
  - Replace the root key with the key of the last node w
  - Delete w
  - Restore the heap-order property (discussed next)



```
ItemType remove(Heap *H) {
NodeType  L;                    /* let L be the last node of H in level order */
NodeType  R;                    /* R is used refer to the root node of H */
ItemType  ItemToRemove;         /* temporarily stores item to remove */
     if  (H is not empty)  {
          /* Remove the highest priority item which is stored in H's root node, R */
          ItemToRemove = (the value stored in the root node, R, of H);
          /* Move L's value into the root of H, and delete L */
          (R's value) = (the value in last node L);
          (delete node L);
          /* Reheapify the values in the remaining nodes of H starting at the root, R */
          if  (H is not empty)  {
             (Reheapify the heap H starting at node R);
          }
  |       return (ItemToRemove);
     }
}
```

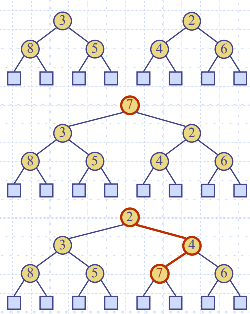

Downheap

- After replacing the root key with the key k of the last node, the heap-order property may be violated
- Algorithm downheap restores the heap-order property by swapping key k along a downward path from the root
- Downheap terminates when key k reaches a leaf or a node whose children have keys greater than or equal to k
- Since a heap has height $O(\log n)$, downheap runs in $O(\log n)$ time



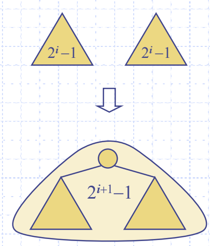
Merging Two Heaps

- We are given two heaps and a key k
- We create a new heap with the root node storing k and with the two heaps as subtrees
- We perform downheap to restore the heap-order property



Bottom-up Heap Construction

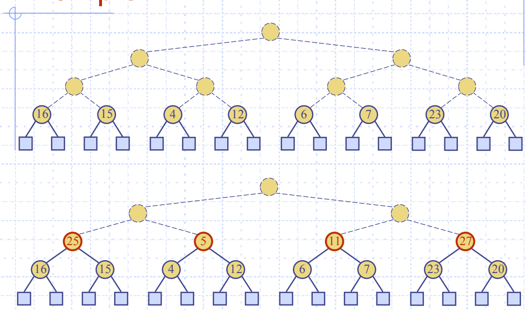
- We can construct a heap storing n keys by using a bottom-up construction with $\log n$ phases
- In phase i, pairs of heaps with 2^i-1 keys are merged into heaps with $2^{i+1}-1$ keys



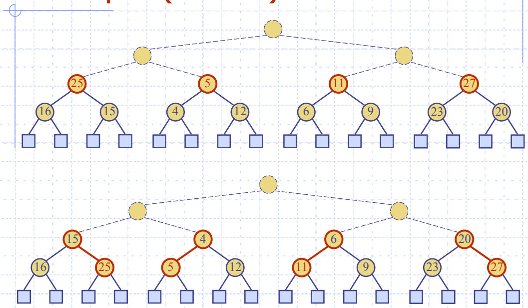
Bottom-up Algorithm:

```
function BottomUpHeap(S, n):
  Input: An array S storing n = 2^h – 1 keys
  Output: A heap T storing the keys in S
  if (n == 0) then return NULL;
  k = S[0];
  S1 = S[1 .. (n-1)/2];       /* split S into two sub-arrays */
  S2 = S[(n-1)/2+1 .. n];
  T1 = BottomUpHeap(S1, (n-1)/2);
  T2 = BottomUpHeap(S2, (n-1)/2);
  T = treeNode(k, T1, T2);
  DownHeap(T);
  return T;
```

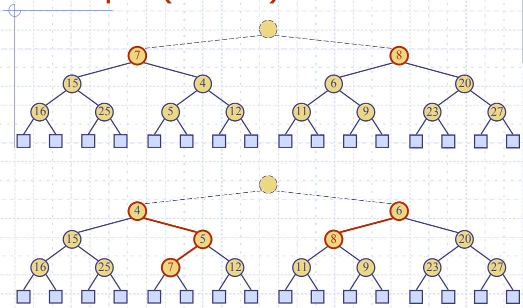

Example



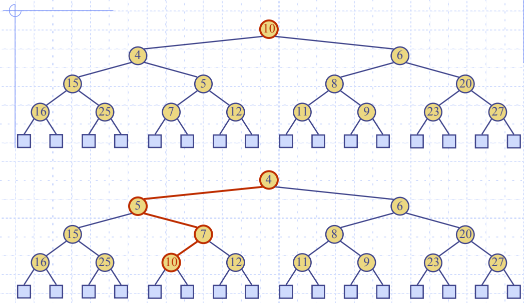
Example (contd.)



Example (contd.)



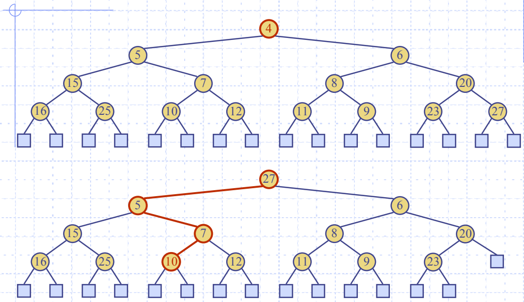
Example (end)



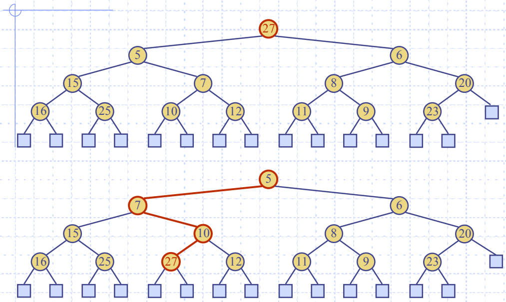
Heapsort

- Heapsort is based on heap. It is a two-stage algorithm.
- The first stage is heap construction. A heap is constructed from the items to be sorted. This stage is $O(n)$
- The second stage is minimum removals. Each time the minimum at the heap root is removed and placed in the result, then the downheap algorithm is used to restore the heap-order property. This process continues until all the items have been placed in the result. This stage is $O(n \log n)$.

Heapsort Example



Heapsort Example (cont.)

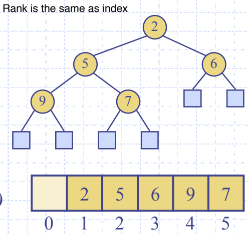


Array-based Implementation

If we use array implementation we don't need a pointer to its children

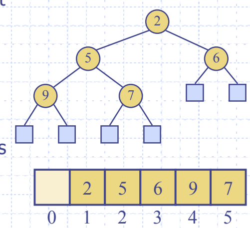
Rank is the same as index

- We can represent a heap with n keys by means of an array of length $n + 1$
- For the node at rank i
  - the left child is at rank $2i$
  - the right child is at rank $2i + 1$
  - Parent is at rank floor($i/2$)
  - Parental nodes are in the first floor($i/2$) ranks.



Array-based Implementation

- Links between nodes are not explicitly stored
- The leaves are not represented
- The cell of at rank 0 is not used
- Operation insert corresponds to inserting at rank $n + 1$
- Operation remove corresponds to removing at rank 1



We don't have to create a tree with nodes
We can do the same operation within an array

Heapsort example

Sort the list 1, 9, 7, 4, 3, 8 in ascending order by heapsort

Stage 1 (heap construction)

1 9 7 4 3 8
1 9 8 4 3 7
1 9 8 4 3 7
9 1 8 4 3 7
9 4 8 1 3 7
1 2 3 4 5 6

Compare 7(pos 3) with 8(child)
and we swap

e
x
Down heap with 9, compare it with its two children
9 is index 2, so children are 4 and 5 (4,3)
9 is already greater than both of them so we dont have to swap
Next parental node, 1 is at position 1 so its two children are at position 2 and 3
its less than both 9 and 8 so we swap it with the bigger one

We then have to downheap one again

Stage 2 (root/max removal)

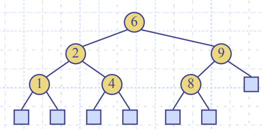
9 4 8 1 3 7
7 4 8 1 3 | 9
8 4 7 1 3 | 9
3 4 7 1 | 8 9
7 4 3 1 | 8 9
1 4 3 | 7 8 9
4 1 3 | 7 8 9
3 1 | 4 7 8 9
3 1 | 4 7 8 9
1 | 3 4 7 8 9
1 2 3 4 5 6

Biggest value and swap with the last value so we swap it with seven. The vertical bar is to separate the sorted and unsorted. Then the second largest value (8) comes to the first position, then we swap it with the last value in the unsorted list.

Binary Search Tree

Satisfys a condition, at any parental node, its key value is >= all the keys in the left subtree and its key value <= all the keys in the right subtree

- A binary search tree is a binary tree storing keys (or key-element pairs) at its internal nodes and satisfying the following property:
  - Let v be a tree node, and L, R be subtrees such that L is the left subtree of v and R is the right subtree of v. We have
    $keys(L) \leq key(v) \leq keys(R)$
- External nodes do not store items (NULL's)
- An inorder traversal of a binary search trees visits the keys in increasing order

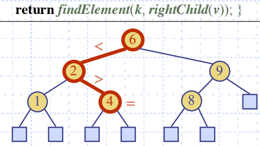


Search

What the find, and the pointer to the head
If the key is less than the key of the node,
then we go to left child, if the target is greater than the key
then we go to the right subtree

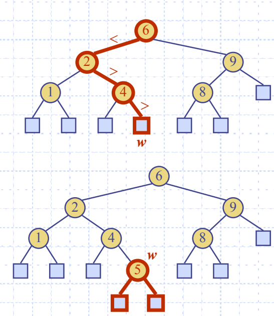
- To search for a key k, we trace a downward path starting at the root
- The next node visited depends on the outcome of the comparison of k with the key of the current node
- If we reach a leaf, the key is not found and we return NO_SUCH_KEY
- Example: findElement(4, root)

```
function findElement(k, v)
    if (isExternal (v)){
        return NO_SUCH_KEY; }
    if (k < key(v)) {
        return findElement(k, leftChild(v));}
    else if (k = key(v)){
        return element(v); }
    else { // k > key(v)
        return findElement(k, rightChild(v)); }
```



Insertion

- To perform operation insertItem(k, o), we search for key k
- Assume k is not already in the tree, and let w be the leaf reached by the search
- We insert k at node w and expand w into an internal node
- Example: insert 5

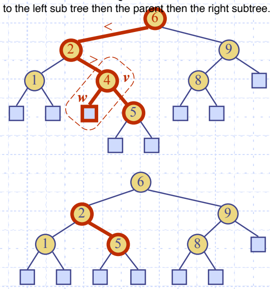


7

2 situations, when we need to delete a node
We first use the search method to find it, it it's not in the tree we don't need to find anything
when we want to delete 4, we start at the root, 4 < 6, go left, 4 > 2 go right, the node is what we want to delete.
Before we remove the node from the tree, we have to check the node. we have to check if it has internal nodes as children.
0 1 or 2 if it has 0 internal nodes. (If we wanted to remove 5 we can just delete and remove it and remove, and replaced with an external node. However if it has one internal node like 4, it has an external node and an internal node.
we have to do some type of reconnection, we need to connect its only child with its parent. Third case is if the node to be deleted has both left and right children and both are internal nodes
If we want to delete 3 >>>>>> 3 has both left and right child. We will find the next node of it in order (in order traversal) (we go to a node and visit the left child and the right child. We find the NEXT node of the node we want to delete we go 1 2 3 5 since we go to the left sub tree then the parent then the right subtree. We remove the node we want to delete and replace it with the node after it.

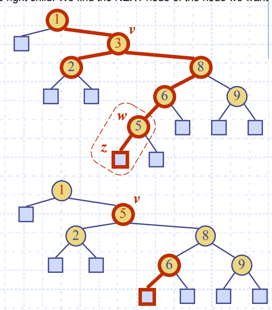
Deletion

- To perform operation removeElement(k), we search for key k
- Assume key k is in the tree, and let v be the node storing k
- If node v has a leaf child w (a NULL subtree), we remove v and w from the tree with operation removeAboveExternal(w)
- Example: remove 4



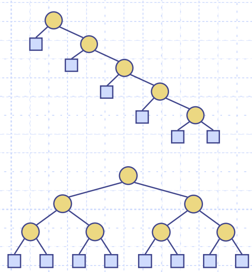
Deletion (cont.)

- We consider the case where the key k to be removed is stored at a node v whose children are both internal
  - we find the internal node w that follows v in an inorder traversal
  - we copy key(w) into node v
  - we remove node w and its left child z (which must be a leaf) by means of operation removeAboveExternal(z)
- Example: remove 3



Performance

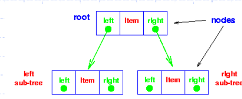
- Consider a dictionary with n items implemented by means of a binary search tree of height h
  - the space used is $O(n)$
  - methods findElement , insertItem and removeElement take $O(h)$ time
- The height h is $O(n)$ in the worst case and $O(\log n)$ in the best case



Trees - Implementation

At least 3 data members, 1 element of data and then a pointer to the left node and a pointer to the right node.
We can have another struct for the root which contains the head and maybe other data members such as how many nodes1

- Data structure




```
typedef struct node{
    void *item;
    struct node *left;
    struct node *right;
} node;

Typedef struct {
    node* root;
    ……
} tree;
```


Trees - Implementation

```
extern int keyCmp( void *a, void *b );
/* Returns -1, 0, 1 for a < b, a == b, a > b */

void *find( node* np, void *key ) {
    if ( np == NULL) return NULL;
    switch( keyCmp( key, np->item) ) {
        case -1 : return find( np->left, key );
        case 0:   return np->item;
        case +1 : return find( np->right, key );
        }
    }

void *findInTree( tree t, void *key ) {
    return find( t.root, key );
    }
```

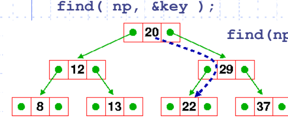
Less, search left

Greater, search right

Trees - Implementation

- Example:
  - key = 22;
    if (findInTree(t.root , &key)) ….

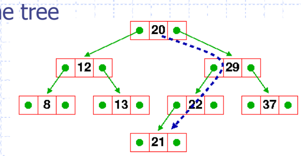
```
find( np, &key );
```




```
find(np->right,&key );
find(np->left,&key );
return np->item;
```

Trees - Addition

- Add 21 to the tree



- We need at most $h+1$ comparisons
- Create a new node (constant time)
- So addition to a tree takes time proportional to $\log n$

Trees - Addition - implementation

```
void insert( node **t, node *new ) {
   node base = *t;
   if ( base == NULL ) {
      *t = new; return; }
   else {
      if( keyLess(new->item, base->item) )
         insert( &(base->left), new );
      else
         insert( &(base->right), new );
   }
}

void addToTree( tree t, void *item ) {
        node* new;
        new = (node*) malloc(sizeof(struct t_node));
        new->item = item;
        new->left = new->right = NULL;
        insert( &(t.root), new );
}
```