DATA STRUCTURES AND ALGORITHMS
LECTURE 11

Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2017

In Lecture 10...

- Hash tables
  - Coalesced chaining
  - Open Addressing
  - Perfect hashing
  - Cuckoo hashing

Today

1. Hash tables
2. Trees

Open addressing - linear probing - reminder

- In case of open addressing all the elements are stored in the table.
- For linear probing the hash function is the following:

$$h(k, i) = (h'(k) + i) \ mod \ m \ \ \forall i = 0, ..., m-1$$

- where $h'(k)$ is a *simple* hash function (for example: $h'(k) = k \ mod \ m$)
- the *probe sequence* for linear probing is: $< h'(k), h'(k)+1, h'(k)+2, ..., m-1, 0, 1, ..., h'(k)-1 >$

Open addressing - linear probing - reminder

- The main disadvantage of linear probing is *primary (and secondary) clustering* - long sequences of occupied slots which tend to grow even longer.
- Clusters are a problem, because unsuccessful search has to check locations until an empty one is found.
- An advantage of linear probing is that consecutive positions have to be checked.

Hopscotch hashing

- *Hopscotch* hashing tries to improve the complexity of the search algorithm for linear probing.
- The main idea of hopscotch hashing is to define for each position from the hash table a neighborhood of size H (H is constant) and to make sure that every element that hashes to a position i will be placed somewhere in the neighborhood of position i.
- Thus, when we search for an element, we will have to check only the neighborhood of the position to which the element hashes.

Hopscotch hashing

- Every position from the hash table has an associated *hop-info* - it is usually a bitmap of size H.
- If in the hop-info of position i, the value of the bit p is set to 1, the element at position $i + p$ hashes to position i.
- If in the hop-info of position i, the value of the bit p is set to 0, the element at position $i + p$ is empty or hashes to a position different from i.

Hopscotch hashing - an example

0		0000
1	67	1010
2	13	1000
3	45	0010
4	92	1010
5	25	0000
6	15	0001
7	51	1000
8	19	1000
9	72	0000
10		0000

h(92, 0) = 4
h(13, 0) = 2
h(51, 0) = 7
h(67, 0) = 1
h(19,0) = 8
h(45, 0) = 1
h(45, 1) = 2
h(45, 2) = 3
h(25, 0) = 3
h(25, 1) = 4
h(25, 2) = 5
h(15, 0) = 4
h(15, 1) = 5
h(25, 2) = 6
h(72, 0) = 6
h(72, 1) = 7
h(72, 2) = 8
h(72, 3) = 9

m = 11
h(k) = k % m
H = 4

- Black lines show the final positions of the elements
- Bold lines show the positions where each element hashes (used to build the hop-info)

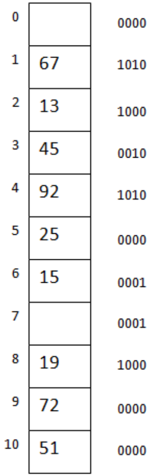
Hopscotch hashing - insert a new element

- When a new element, k, has to be inserted:
    - Compute the initial position, $p = h(k, 0)$ for it
    - If p is empty, we will place k there.
    - If p is not empty, we have to find an empty location, pe, for it (just like regular linear probing).
    - If pe is in the neighborhood of p (between p and $p + H - 1$), we will put it there.
    - If pe is too far, we will try to *move* it closer to p.
    - We will see whether there is an element (closer to p) that can be moved to position pe. If we find such an element, we will move it, and empty its position. Repeat the process until the empty position is close enough so that we can put k there.

Hopscotch hashing - insert example

- Insert the element 90.
- $p = h(90, 0) = 2$
- pe = first empty position $= 10 \Rightarrow$ too far away (neighborhood H $= 4$)
- We want to move something to position 10. Candidates are elements that hash to positions 7, 8, 9 (these have position 10 in their neighborhood).
- We first look at the hop-info of position 7. If it contains a value of 1, we will move that element to position 10, and free up its position.

Hopscotch hashing - insert example



0		0000
1	67	1010
2	13	1000
3	45	0010
4	92	1010
5	25	0000
6	15	0001
7		0001
8	19	1000
9	72	0000
10	51	0000

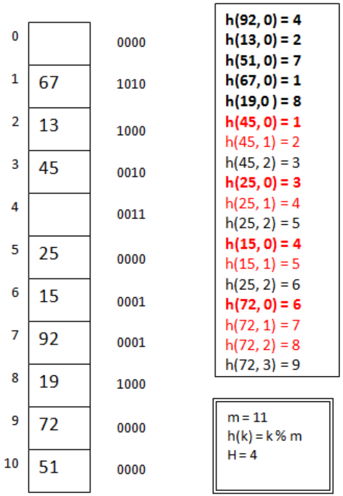
h(92, 0) = 4
h(13, 0) = 2
h(51, 0) = 7
h(67, 0) = 1
h(19,0) = 8
h(45, 0) = 1
h(45, 1) = 2
h(45, 2) = 3
h(25, 0) = 3
h(25, 1) = 4
h(25, 2) = 5
h(15, 0) = 4
h(15, 1) = 5
h(25, 2) = 6
h(72, 0) = 6
h(72, 1) = 7
h(72, 2) = 8
h(72, 3) = 9

m = 11
h(k) = k % m
H = 4

Hopscotch hashing - insert example

- Now position 7 is empty, but it is still too far from position 2 (where element 90 should be put). We try to move something to position 7. Candidates are positions 4, 5, 6.

Hopscotch hashing - insert example



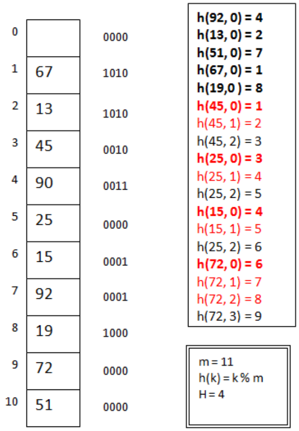
0		0000
1	67	1010
2	13	1000
3	45	0010
4		0011
5	25	0000
6	15	0001
7	92	0001
8	19	1000
9	72	0000
10	51	0000

h(92, 0) = 4
h(13, 0) = 2
h(51, 0) = 7
h(67, 0) = 1
h(19,0) = 8
h(45, 0) = 1
h(45, 1) = 2
h(45, 2) = 3
h(25, 0) = 3
h(25, 1) = 4
h(25, 2) = 5
h(15, 0) = 4
h(15, 1) = 5
h(25, 2) = 6
h(72, 0) = 6
h(72, 1) = 7
h(72, 2) = 8
h(72, 3) = 9

m = 11
h(k) = k % m
H = 4

Hopscotch hashing - insert example

- Position 4 is close enough, we will put 90 there.



Hopscotch hashing - insert

- If no element can be moved to make free space, the element cannot be inserted in the hash table.
- In this situation the table has to be resized and elements have to be inserted again.
- However, Hopscotch hashing handles well a load factor of up to 0.9.

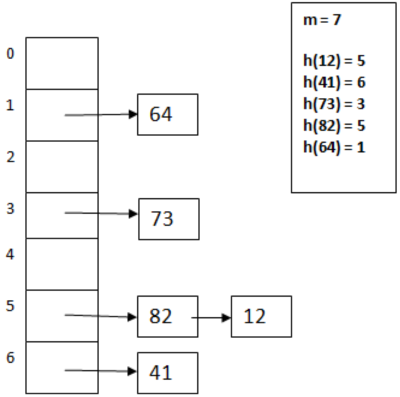
Hopscotch hashing - search and remove

- When we search for an element, we compute the value of the hash function for the element, p, and check the positions corresponding to the 1 bits in the hop-info of p.
- We will have to check at most H locations (and H is constant).
- When we want to remove an element we first search for it (as described above) and if it is found, we simply set its position to empty.

Linked Hash Table

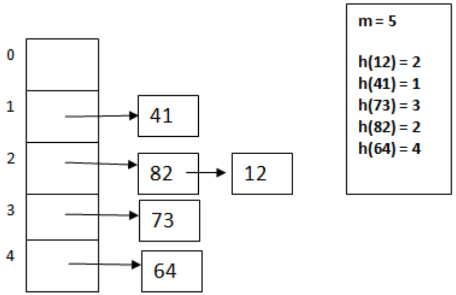
- Assume we build a hash table using separate chaining as a collision resolution method.
- We have discussed how an iterator can be defined for such a hash table.
- When iterating through the elements of a hash table, the order in which the elements are visited is *undefined*
- For example:
  - Assume an initially empty hash table (we do not know its implementation)
  - Insert one-by-one the following elements: 12, 41, 73, 82, 64
  - Use an iterator to display the content of the hash table
  - In what order will the elements be displayed?

Linked Hash Table



- Iteration order: 64, 73, 82, 12, 41

Linked Hash Table



- Iteration order: 41, 82, 12, 73, 64

Linked Hash Table

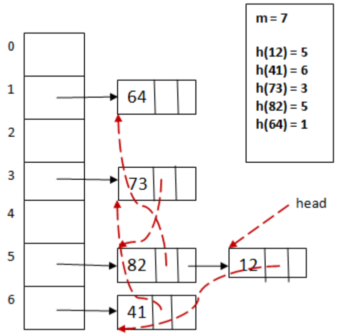
- A *linked hash table* is a data structure which has a *predictable* iteration order. This order is the order in which elements were inserted.
- So if we insert the elements 12, 41, 73, 82, 64 (in this order) in a linked hash table and iterate over the hash table, the iteration order is guaranteed to be: 12, 41, 73, 82, 64.
- How could we implement a linked hash table which provides this iteration order?

Linked Hash Table

- A linked hash table is a combination of a hash table and a linked list. Besides being stored in the hash table, each element is part of a linked list, in which the elements are added in the order in which they are inserted in the table.

- Since it is still a hash table, we want to have, on average, $\Theta(1)$ for insert, remove and search, these are done in the same way as before, the *extra* linked list is used only for iteration.

Linked Hash Table



- Red arrows show how the elements are linked in insertion order, starting from a *head* - the first element that was inserted, 12.

Linked Hash Table

- Do we need a doubly linked list for the order of elements or is a singly linked list sufficient? (think about the operations that we usually have for a hash table).

Linked Hash Table

- Do we need a doubly linked list for the order of elements or is a singly linked list sufficient? (think about the operations that we usually have for a hash table).

- The only operation that cannot be efficiently implemented if we have a singly linked list is the *remove* operation. When we remove an element from a singly linked list we need the element before it, but finding this in our linked hash table takes $O(n)$ time.

Linked Hash Table - Implementation

- What structures do we need to implement a Linked Hash Table?

```
Node:
    info: TKey
    nextH: ↑ Node //pointer to next node from the collision
    nextL: ↑ Node //pointer to next node from the insertion-order list
    prevL: ↑ Node //pointer to prev node from the insertion-order list
```

```
LinkedHT:
    m:Integer
    T:(↑ Node)[]
    h:TFunction
    head: ↑ Node
    tail: ↑ Node
```

Linked Hash Table - Insert

- How can we implement the *insert* operation?

```
subalgorithm insert(lht, k) is:
//pre: lht is a LinkedHT, k is a key
//post: k is added into lht
    allocate(newNode)
    [newNode].info ← k
    @set all pointers of newNode to NIL
    pos ← lht.h(k)
    //first insert newNode into the hash table
    if lht.T[pos] = NIL then
        lht.T[pos] ← newNode
    else
        [newNode].nextH ← lht.T[pos]
        lht.T[pos] ← newNode
    end-if
//continued on the next slide...
```

Linked Hash Table - Insert

```
    //now insert newNode to the end of the insertion-order list
    if lht.head = NIL then
        lht.head ← newNode
        lht.tail ← newNode
    else
        [newNode].prevL ← lht.tail
        [lht.tail].nextL ← newNode
        lht.tail ← newNode
    end-if
end-subalgorithm
```

Linked Hash Table - Remove

- How can we implement the *remove* operation?

```
subalgorithm remove(lht, k) is:
//pre: lht is a LinkedHT, k is a key
//post: k was removed from lht
    pos ← lht.h(k)
    current ← lht.T[pos]
    nodeToBeRemoved ← NIL
    //first search for k in the collision list and remove it if found
    if current ≠ NIL and [current].info = k then
        nodeToBeRemoved ← current
        lht.T[pos] ← [current].nextH
    else
        prevNode ← NIL
        while current ≠ NIL and [current].info ≠ k execute
            prevNode ← current
            current ← [current].nextH
        end-while
//continued on the next slide...
```

```
        if current $\neq$ NIL then
            nodeToBeRemoved $\leftarrow$ current
            [prevNode].nextH $\leftarrow$ [current].nextH
        else
            @k is not in lht
        end-if
    end-if
//if k was in lht then nodeToBeRemoved is the address of the node containing
//it and the node was already removed from the collision list - we need to
//remove it from the insertion-order list as well
    if nodeToBeRemoved $\neq$ NIL then
        if nodeToBeRemoved = lht.head then
            if nodeToBeRemoved = lht.tail then
                lht.head $\leftarrow$ NIL
                lht.tail $\leftarrow$ NIL
            else
                lht.head $\leftarrow$ [lht.head].nextL
                [lht.head].prev $\leftarrow$ NIL
            end-if
//continued on the next slide...
```

```
        else if nodeToBeRemoved = lht.tail then
            lht.tail ← [lht.tail].prev
            [lht.tail].next ← NIL
        else
            [[nodeToBeRemoved].next].prev ← [nodeToBeRemoved].prev
            [[nodeToBeRemoved].prev].next ← [nodeToBeRemoved].next
        end-if
        deallocate(nodeToBeRemoved)
    end-if
end-subalgorithm
```

Trees

- Trees are one of the most commonly used data structures because they offer an efficient way of storing data and working with the data.
- In graph theory a *tree* is a connected, acyclic graph (usually undirected).
- When talking about trees as a data structure, we actually mean *rooted trees*, trees in which one node is designated to be the *root* of the tree.

Tree - Definition

- A tree is a finite set $\mathcal{T}$ of 0 or more elements, called *nodes*, with the following properties:
  - If $\mathcal{T}$ is empty, then the tree is empty
  - If $\mathcal{T}$ is not empty then:
    - There is a special node, R, called the *root* of the tree
    - The rest of the nodes are divided into k ($k \geq 0$) disjunct *trees*, T_1, T_2, ..., T_k, the root node R being linked by an edge to the root of each of these trees. The trees T_1, T_2, ..., T_k are called the *subtrees* (*children*) of R, and R is called the *parent* of the subtrees.

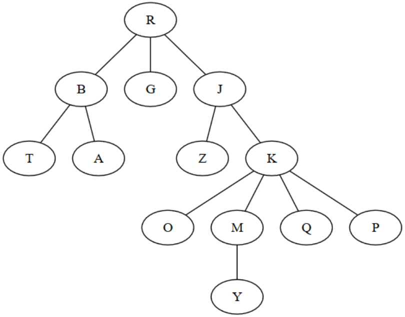
Tree - Terminology I

- An *ordered tree* is a tree in which the order of the children is well defined and relevant (instead of having a set of children, each node has a list of children).
- The *degree* of a node is defined as the number of children of the node.
- The nodes with the degree 0 (nodes without children) are called *leaf nodes*.
- The nodes that are not leaf nodes are called *internal nodes*.

Tree - Terminology II

- The *depth* or *level* of a node is the length of the path (measured as the number of edges traversed) from the root to the node. This path is unique. The root of the tree is at level 0 (and has depth 0).
- The *height* of a node is the length of the longest path from the node to a leaf node.
- The *height of the tree* is defined as the height of the root node, i.e., the length of the longest path from the root to a leaf.

Tree - Terminology Example



- Root of the tree: R
- Children of R: B, G, J
- Parent of M: K
- Leaf nodes: T, A, G, Z, O, Y, Q, P
- Internal nodes: R, B, J, K, M
- Depth of node K: 2 (path R-J-K)
- Height of node K: 2 (path K-M-Y)
- Height of the tree (height of node R): 4
- Nodes on level 2: T, A, Z, K

k-ary trees

- How can we represent a tree in which every node has at most k children?
- One option is to have a structure for a *node* that contains the following:
  - information from the node
  - address of the parent node (not mandatory)
  - k fields, one for each child

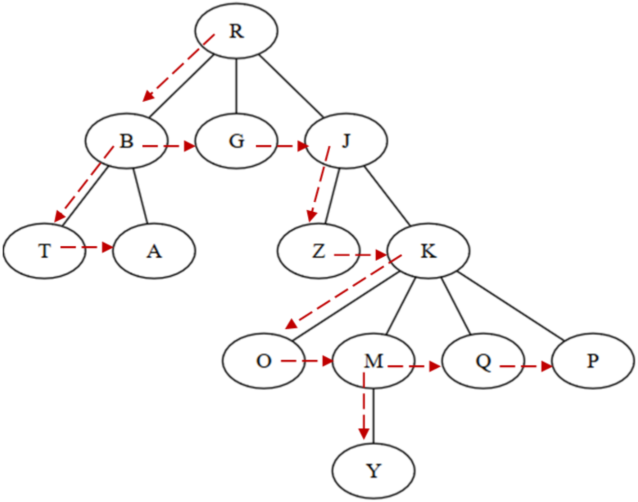
k-ary trees

- Another option is to have a structure for a *node* that contains the following:
    - information from the node
    - address of the parent node (not mandatory)
    - an array of dimension k, in which each element is the address of a child
    - number of children (number of occupied positions from the above array)
- Disadvantage of these approaches is that we occupy space for k children even if most nodes have less children.

k-ary trees

- A third option is the so-called *left-child right-sibling* representation in which we have a structure for a node which contains the following:
  - information from the node
  - address of the parent node (not mandatory)
  - address of the leftmost child of the node
  - address of the right sibling of the node (next node on the same level).

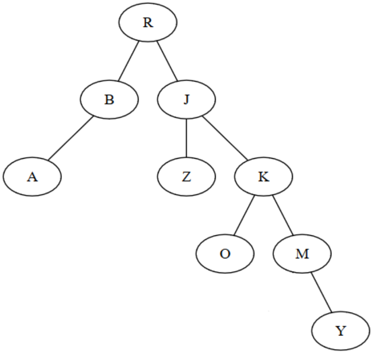
Left-child right sibling representation example



Binary trees

- An ordered tree in which each node has at most two children is called *binary tree*.
- In a binary tree we call the children of a node the *left child* and *right child*.
- Even if a node has only one child, we still have to know whether that is the left or the right one.

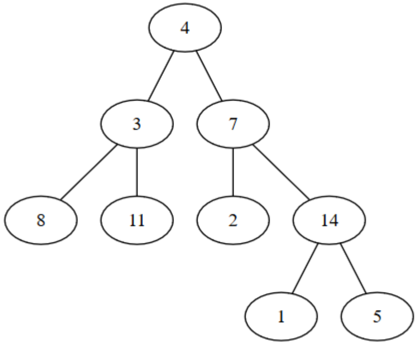
Binary tree - example



- A is the left child of B
- Y is the right child of M

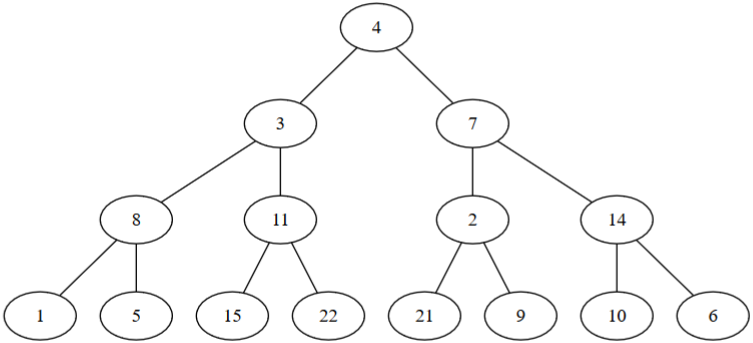
Binary tree - Terminology I

- A binary tree is called *full* if every internal node has exactly two children.



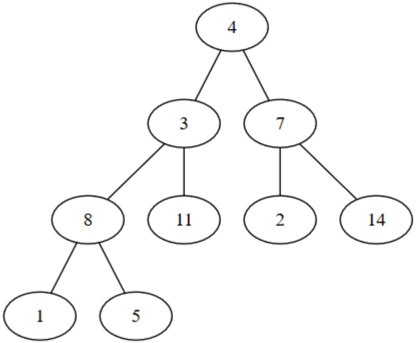
Binary tree - Terminology II

- A binary tree is called *complete* if all leaves are one the same level and all internal nodes have exactly 2 children.



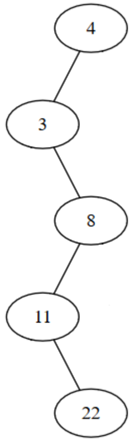
Binary tree - Terminology III

- A binary tree is called *almost complete* if it is a *complete* binary tree except for the last level, where nodes are completed from left to right (binary heap - structure).



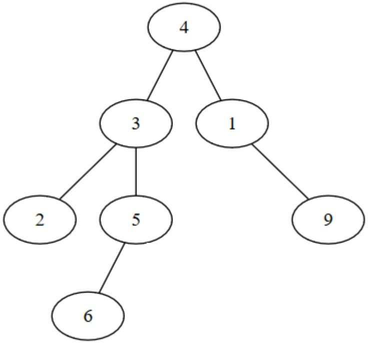
Binary tree - Terminology IV

- A binary tree is called *degenerate* if every internal node has exactly one child (it is actually a chain of nodes).



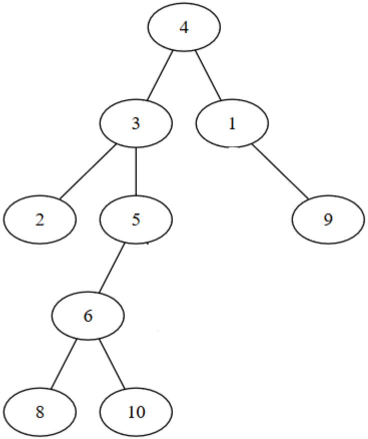
Binary tree - Terminology V

- A binary tree is called *balanced* if the difference between the height of the left and right subtrees is at most 1 (for every node from the tree).



Binary tree - Terminology VI

- Obviously, there are many binary trees that are none of the above categories, for example:



Binary tree - properties

- A binary tree with n nodes has exactly $n - 1$ edges (this is true for every tree, not just binary trees)
- The number of nodes in a complete binary tree of height N is $2^{N+1} - 1$ (it is $1 + 2 + 4 + 8 + ... + 2^N$)
- The maximum number of nodes in a binary tree of height N is $2^{N+1} - 1$ - if the tree is complete.
- The minimum number of nodes in a binary tree of height N is N - if the tree is degenerate.
- A binary tree with N nodes has a height between $log_2 N$ and N.

ADT Binary Tree I

- Domain of ADT Binary Tree:

$$\mathcal{BT} = \{bt \mid bt \text{ binary tree with nodes containing information of type TElem}\}$$

ADT Binary Tree II

- init(bt)
  - **descr:** creates a new, empty binary tree
  - **pre:** true
  - **post:** $bt \in \mathcal{BT}$, bt is an empty binary tree

ADT Binary Tree III

- initLeaf(bt, e)
  - **descr:** creates a new binary tree, having only the root with a given value
  - **pre:** $e \in TElem$
  - **post:** $bt \in \mathcal{BT}$, bt is a binary tree with only one node (its root) which contains the value e

ADT Binary Tree IV

- initTree(bt, left, e, right)
    - **descr:** creates a new binary tree, having a given information in the root and two given binary trees as children
    - **pre:** $left, right \in \mathcal{BT}$, $e \in TElem$
    - **post:** $bt \in \mathcal{BT}$, *bt* is a binary tree with left child equal to *left*, right child equal to *right* and the information from the root is *e*

ADT Binary Tree V

- insertLeftSubtree(bt, left)
    - **descr:** sets the left subtree of a binary tree to a given value (if the tree had a left subtree, it will be changed)
    - **pre:** $bt, left \in \mathcal{BT}$
    - **post:** $bt' \in \mathcal{BT}$, the left subtree of bt' is equal to *left*

ADT Binary Tree VI

- insertRightSubtree(bt, right)
    - **descr:** sets the right subtree of a binary tree to a given value (if the tree had a right subtree, it will be changed)
    - **pre:** $bt, right \in \mathcal{BT}$
    - **post:** $bt' \in \mathcal{BT}$, the right subtree of bt' is equal to *right*

ADT Binary Tree VII

- root(bt)
  - **descr:** returns the information from the root of a binary tree
  - **pre:** $bt \in \mathcal{BT}$, $bt \neq \Phi$
  - **post:** $root \leftarrow e$, $e \in TElem$, e is the information from the root of bt
  - **throws:** an exception if bt is empty

ADT Binary Tree VIII

- left(bt)
  - **descr:** returns the left subtree of a binary tree
  - **pre:** $bt \in \mathcal{BT}$, $bt \neq \Phi$
  - **post:** $left \leftarrow, l$, $l \in \mathcal{BT}$, l is the left subtree of bt
  - **throws:** an exception if bt is empty

ADT Binary Tree IX

- right(bt)
  - **descr:** returns the right subtree of a binary tree
  - **pre:** $bt \in \mathcal{BT}$, $bt \neq \Phi$
  - **post:** $right \leftarrow r$, $r \in \mathcal{BT}$, r is the right subtree of bt
  - **throws:** an exception if bt is empty

ADT Binary Tree X

- isEmpty(bt)
    - **descr:** checks if a binary tree is empty
    - **pre:** $bt \in \mathcal{BT}$
    - **post:**

$$empty \leftarrow \begin{cases} True, & \text{if } bt = \Phi \\ False, & \text{otherwise} \end{cases}$$

ADT Binary Tree XI

- iterator (bt, traversal, i)
  - **descr:** returns an iterator for a binary tree
  - **pre:** $bt \in \mathcal{BT}$, *traversal* represents the order in which the tree has to be traversed
  - **post:** $i \in \mathcal{I}$, *i* is an iterator over *bt* that iterates in the order given by *traversal*

ADT Binary Tree XII

- destroy(bt)
  - **descr:** destorys a binary tree
  - **pre:** $bt \in \mathcal{BT}$
  - **post:** bt was destroyed

ADT Binary Tree XIII

- Other possible operations:
  - change the information from the root of a binary tree
  - remove a subtree (left or right) of a binary tree
  - search for an element in a binary tree
  - return the number of elements from a binary tree

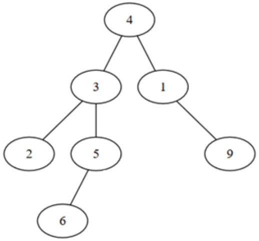
Possible representations

- If we want to implement a binary tree, what representation can we use?
- We have several options:
    - Representation using an array (similar to a binary heap)
    - Linked representation
        - with dynamic allocation
        - on an array

Possible representations I

- Representation using an array
  - Store the elements in an array
  - First position from the array is the root of the tree
  - Left child of node from position i is at position $2 * i$, right child is at position $2 * i + 1$.
  - Some special value is needed to denote the place where no element is.

Possible representations II



Pos	Elem
1	4
2	3
3	1
4	2
5	5
6	-1
7	9
8	-1
9	-1
10	6
11	-1
12	-1
13	-1
...	...

- Disadvantage: depending on the form of the tree, we might waste a lot of space.

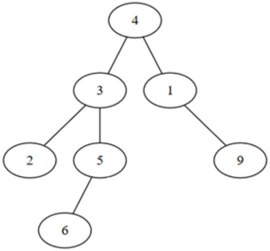
Possible representations I

- Linked representation with dynamic allocation
  - We have a structure to represent a node, containing the information, the address of the left child and the address of the right child (possibly the address of the parent as well).
  - An empty tree is denoted by the value NIL for the root.
  - We have one node for every element of the tree.

Possible representations II

- Linked representation on an array
  - Information from the nodes is placed in an array. The *address* of the left and right child is the *index* where the corresponding elements can be found in the array.
  - We can have a separate array for the parent as well.

Possible representations III



Pos	1	2	3	4	5	6	7	8
Info	4	3	2	5	6	1	9	
Left	2	3	-1	5	-1	-1	-1	
Right	6	4	-1	-1	-1	7	-1	
Parent	-1	1	2	2	4	1	6	

- We need to know that the root is at position 1 (could be any position).
- If the array is full, we can allocate a larger one.
- We can keep a linked list of empty positions to make adding a new node easier.